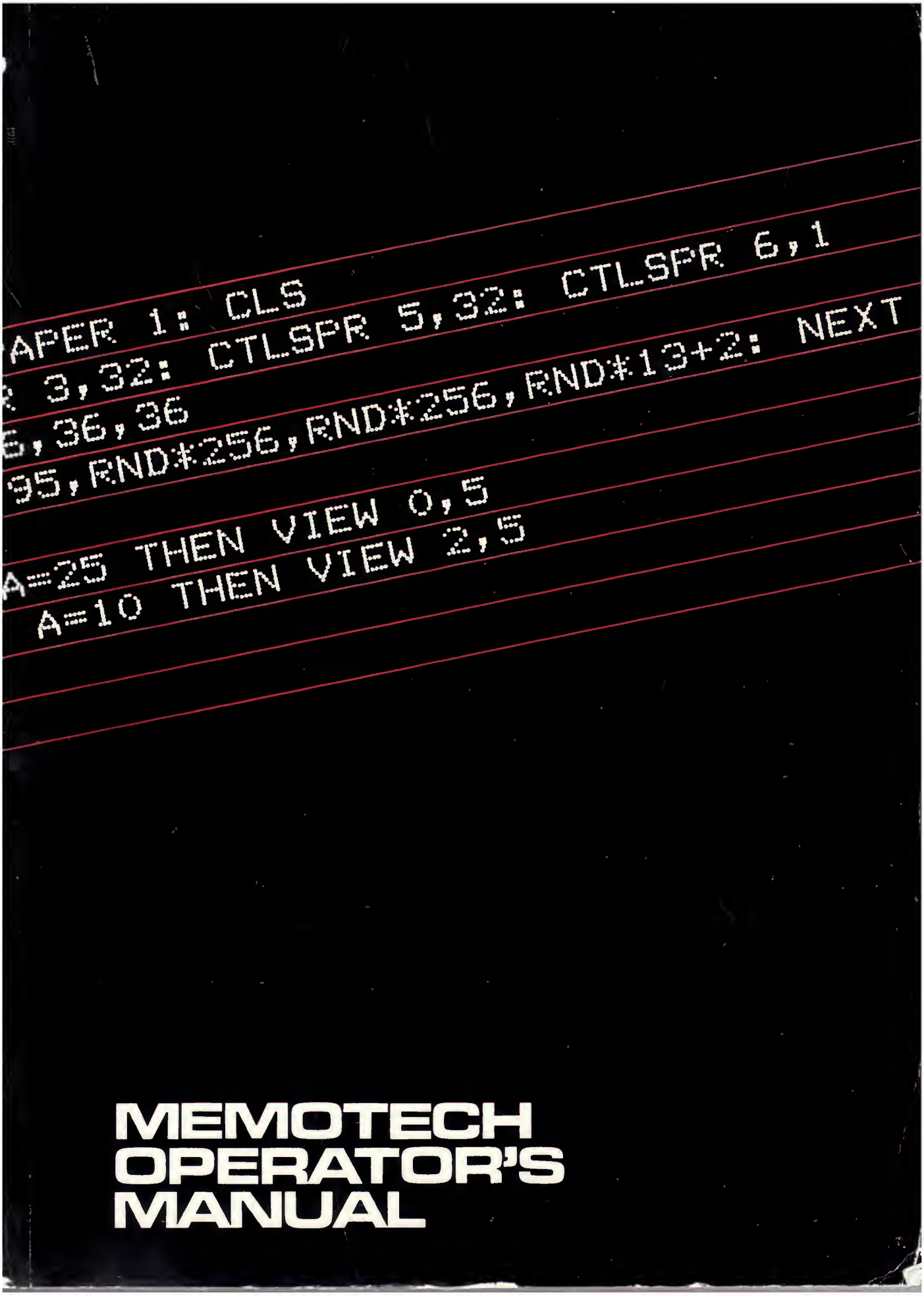


MEMOTECH OPERATOR'S MANUAL

THE MEMOTECH

Operator's Manual

Written by
Spencer Bateson

Published for Memotech Ltd.
by

PHOENIX PUBLISHING ASSOCIATES LTD.
14 VERNON ROAD, BUSHEY, HERTS. WD2 2JL

ISBN 0 9465 7626 2
Copyright © Spencer Bateson
Typeset by First Page Ltd, Watford
Designed by Denis Gibney Graphics, Chesham
Printed by The Garden City Press, Letchworth.

CONTENTS

NB. In this book you will see " " ' ' used for quotation marks. This is a feature of the type used in producing this book. Your computer key however will show .

CHAPTER 1 : THE SET-UP

PACKAGES AND PERIPHERALS

Welcome to the world of MTX computing! Along with the computer itself you should find the following goodies in the Memotech package.

1. One Memotech power pack
2. One cassette lead
3. One TV lead
4. One snap in cartridge port cover (possibly attached)
5. Five complimentary cassette programs

In a moment we'll look at each of these peripherals in turn, explaining what each piece of equipment does and, when relevant, how it should be connected to your computer.

Before you set up your new acquisition it is a good idea to give some thought to the type of work surface required. Obviously it should be near a power supply and large enough to comfortably hold the Memotech, a TV and a cassette player (if you have one). If it is possible try to find somewhere that will serve as a permanent computer work station. Apart from the obvious considerations of convenience, a dedicated base of this type ensures that there is less chance of damaging the Memotech's ports and leads by constantly plugging in and disconnecting your peripherals. Unsurprisingly, your Memotech needs to be fed electricity before it can perform any wondrous feats! This is achieved by inserting the MTX lead (or power pack as its known in the trade) into the socket marked 'POWER' on the back of the computer. You will then need to flick the switch on the side of the power pack which should light up, thus indicating that the Memotech is 'powered up'.

As far as the micro is concerned it's now ready and rearing to go, but since we are only mere humans and seeing is believing, we need the computer to display information in a manner that we can understand. This is achieved by the use of a Visual Display Unit (VDU), which for the majority of you will mean a TV. Before we explain how to connect your VDU to the Memotech a few words of advice are in order.

We have already stressed the value of a dedicated work station, but how can this be achieved without a VDU which is a permanent part of the system? Well, the fact of the matter is that it can't, which is why it's advisable to get your hands on a TV exclusively dedicated to computing. Apart from the considerations previously discussed there are a number of reasons that this somewhat luxurious situation is desirable.

Aside from the obvious advantages of avoiding conflict with other members of the family who are more interested in soap opera than computing, there are other slightly more serious considerations. The aerial socket at the back of a TV was not designed to have plugs constantly pushed in and pulled out of it. With a little care (and a total absence of brute force), damage to connectors can probably be avoided.

Right, let's set up the basic MTX system. Fish out the TV lead and plug the appropriate connection into the aerial socket on the TV and attach the other to the socket marked 'TV' on the back of the Memotech (it is easy to tell which end should be plugged into which piece of equipment). Having achieved the rudiments of a system you need to tune the TV into channel 36 (the channel used by most micros). When the MTX is correctly tuned in the following message should be displayed on a clear blue background (that is providing you have a colour TV!):

```
■
Ready
```

This message is displayed whenever you turn on your micro. This is the computer's way of telling you that it's awaiting instructions. There should also be a flashing white square above the 'Ready' prompt which is known as a cursor. The cursor indicates the position at which characters will be printed to the screen.

The first component you should consider adding to your system is a cassette player. Whilst, theoretically, this is an optional extra, in practice computing life is impossible without one. As you probably know, cassette recorders provide computers with 'offline' storage (excuse the jargon!). Such a facility is referred to as 'offline' simply because it is a storage facility external to the computer itself. In essence, this means that you can store the program/file currently held in the micro's memory on a tape, reloading it as and when required. Cassette handling is discussed at some length in chapter 5, so we won't go into detail at this stage but merely outline a few essential points.

The MTX cassette Head Cleaner is a very valuable addition to your system which is likely to be overlooked. If the heads on your cassette player get covered in magnetic dust or assorted grime it becomes impossible to reliably save and load programs. This is more than a little frustrating. Imagine spending hours meticulously creating an earth shattering program, storing it on a cassette, only to come back at a later date to discover that you are unable to reload the program because you've neglected to keep the heads clean. The moral of this rather stern lecture is clear: Developing a program takes time and trouble. If you want to avoid the disappointment of losing a cherished creation you must get into the habit of using the head cleaner on a regular basis!

If you don't yet own a cassette recorder, resist a rash purchase until you've read chapter 5 of this manual. This will provide you with all the essential parameters you need in order to select the most appropriate type of recorder for the task at hand. For now, all you need to bear in mind is that you don't need to spend a fortune to equip yourself with a perfectly adequate data recorder.

The next item on our peripheral check list is the snap-in cartridge port cover. It is quite feasible that you won't be able to find this item when you open up the MTX package. Since it's often attached to the computer on delivery. Take a look at the left-hand side of your computer and locate the cartridge port to be protected by this cover. When this port is not in use it is advisable to plug in the cartridge port cover to prevent pollution from dust and grime. So if the cover is not attached, clip it into place.

Having pieced together the basic MTX system, you'll notice a few ports on the back of the Memotech that have not yet been mentioned. These sockets offer a wealth of expansion possibilities, so we'll take a quick look at what you computing future holds.

On the left of the computer you'll see two ports labelled RS232-1 and RS232-0. These can't be used in their 'raw' state. However, with the addition of further peripherals you can use a serial printer and disc drive as well as exploiting the MTX's networking potential.

On the right-hand side of your micro you'll find a port labelled CENTRONICS TYPE PARALLEL PRINTER, which (predictably enough) enables MTX programmers to utilise a parallel printer. Your micro's Input/ Output (I/O) facilities are unusually flexible in that they allow access to both centronics and serial printers.

Next to the RS232-0 port is a socket marked 'MONITOR'. A monitor is another type of VDU which can be used instead of a TV. The main difference between the two options lies in the the quality of the screen display, which the monitor wins hands down. Monitors are simply high quality VDU's which offer a much sharper display than TV's. The majority of monitors are very expensive and are generally found dedicated to a computer system. This said, if you want the best of both worlds, there is a wide range of TV/monitors available which in terms, of both quality and price, offer an acceptable compromise for the home user. However, if you decide upon this compromise solution, ensure that the VDU you opt for is fitted with a BNC connector.

The last 'add-on' we'll outline in this chapter is the disc drive. To use the MTX with CP/M you'll require 64K of memory. A single disc system can be used by either 32K or 64K but an additional board, known as the Communications Board. is required by both systems. A disc drive can be used as an alternative means of storing your files and programs. It plays exactly the same role as a cassette recorder, but it is more reliable, substantially faster and operates far more efficiently.

The MTX drive scans a disc for a specified file, which is then loaded into the computer's memory as soon as it is found. The wonder of disc drives is that they only take a few seconds to load and save programs. Although this is a fairly expensive method of storage, the majority of serious users insist that, in the final analysis. the purchase of a cassette recorder is simply delaying the inevitable - a disc drive is the essential add-on!

Having said that a disc drive with CP/M can only be operated with a 64K system, this is probably the time to explain what 64K and 32K actually mean. The MTX is equipped with either 32K or 64K of available memory and since CP/M consumes a fair amount of this valuable memory space it can't operate with a 32K system. For the moment, all you need to know is that a 'byte' is a storage location in your micro's memory and that K (or kilobyte) is an expression used to denote the amount of available memory. One kilobyte will hold approximately one page of this book, and so, 32K (kilobytes) will hold approximately 32 pages.

By now you and your micro are presumably powered-up and ready to go. In the chapters which follow we'll introduce you to the BASIC computer language, and in so doing ensure that you and your computer travel in the same direction.

CHAPTER 2 : BASIC COMMUNICATIONS

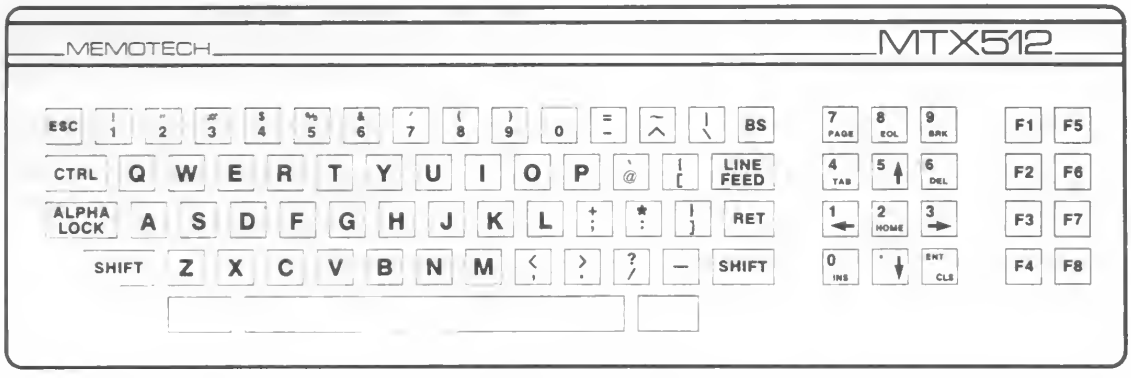
WHAT'S BASIC

The computer's humming and your fingers are poised over the keyboard, but how do you get the silicon wonder to do anything? Well, the first thing to recognise is that although micros can offer fairly convincing simulations of intelligence, they're not actually very bright! In fact computers must be given very precise instructions before they are capable of anything. In the next few chapters we will be looking at one of the languages that can be used to persuade your micro to perform whatever fiendish tasks you choose to set it.

There are three main methods of communicating with your MTX - Noddy, Assembler and BASIC. The language that we'll be looking at in the next few chapters is BASIC (Beginners All-Purpose Symbolic Instruction Code). The BASIC language is known as a high-level language. This means that we don't have to concern ourselves with the complex clicks and whirs going on within the computer, all we have to do is issue instructions, according to the rules of BASIC, and watch the computer carry them out.

It is imperative that you don't skip any of this section since it will provide you with the foundations of the BASIC language. Work your way through the text systematically, trying all the examples as you go and in this way you will gain an understanding of the intricacies of BASIC.

The Keyboard



The keys on the Memotech keyboard are laid out in the standard QWERTY typewriter format, so if you are accustomed to using a typewriter you won't find the MTX a problem. Try typing in a few characters, that is, numbers, letters and any of the symbols on the keys and then press <RET> key. You will probably be confronted with a screen displaying the characters you typed in, with the cursor flashing over the first character of the line, and a message that says 'Mistake'. This will all be explained later in the chapter, but for now simply note computers are very meticulous about the information they will accept and this message is the computer's way of telling you that it doesn't understand what you have just typed in. To get rid of the error message press the CLS key (bottom right of the numeric keypad on the right of the machine) and then press <RET> and the 'Ready' prompt will reappear.

It is important to realise that nothing you can type in can actually hurt the machine. If anything apparently disastrous should happen - such as the keyboard not responding when you press the keys - either switch the MTX off and on again or simultaneously press the two keys on either side of the space bar and the keyboard response will return. This process is known (rather unimaginatively) as RESETting the machine and consequently the keys are called the RESET keys.

Up until now all the letters that you have entered have probably been in upper case (capitals), to access the lower case set of characters press the ALPHA LOCK key. You will see that all the characters that you type in are now in lower case. To return to the upper case mode you can either press the ALPHA LOCK key again or else use the SHIFT key (as on a typewriter) to access single characters. We'll now take a look at the non-standard keys on the MTX keyboard.

ALPHA LOCK

This key, situated on the left-hand side of the QWERTY keyboard, acts as a toggle switch from upper case to lower case. A toggle acts in much the same way as a light switch turning a facility on or off.

SHIFT

The SHIFT keys (which you will find bottom left and right of the QWERTY keyboard) provide access to both capitals (assuming you were in lower case) and the top symbol on the key that is required. For example if you press SHIFT and 1 simultaneously an exclamation mark (!) will be printed to the screen.

CTRL

The control key is sometimes used during the running of a program. To give you an example of what it can do try pressing CTRL and 8 together. For additional uses of this facility refer to the Codes of Control appendix.

ESC

Your computer has an unusual feature which enables us to make use of six different character sets. You automatically access the standard international character set (the American set) when you power up your computer and the ESCAPE key is used to access the other sets. Try pressing SHIFT and 3 and, instead of the expected pound sign, a hash (#) appears on the screen. The hash is substituted for the pound sign which is not supported by the American standard set. If you want to use a pound sign you will have to access the English character set, to do this press the ESCAPE key followed by B then 1. A complete list of ESCape sequences is listed in appendix 2.

BS

The BS key, situated on the top right-hand corner of the QWERTY keyboard, is the Back Space key. BS moves the cursor back one character space each time it is pressed. If you hold down the BS key (or any other key for that matter) the key will repeat itself until such time as it is released. This is referred to as an auto repeat facility.

LINE FEED

This key, situated above the <RET> key on the QWERTY keyboard, is mainly used in conjunction with the NODDY language and has the same effect as the cursor down key.

RET

The RET key is situated on the right-hand side of the QWERTY keyboard above the SHIFT key. It has a similar function to the carriage return on a typewriter in that it creates a linefeed whilst returning the cursor to the beginning of the next line. The difference between the the carriage return on a typewriter and the RETurn key on a micro is that the RETurn key will activate a direct command or cause a program line to be stored in the computer's memory as well as performing the standard carriage return function.

THE NUMERIC KEYPAD

The numeric keypad is on the right of the QWERTY keyboard and has two operational modes. By pressing a number and the SHIFT key you can use it to enter numbers and by using it unSHIFTed you can use it to edit your programs.

PAGE

This key (top left of the numeric keypad) is used to pause a program when you are listing it to the screen. It acts as a toggle switch. The first time PAGE is pressed it pauses a program listing, at the same time generating a sound prompt.

EOL

This is the 8 key on the numeric keyboard. EOL is short
for End of Line and, as we have already seen, its function is to delete all characters following the cursor. For further details, see the section on MTX editing facilities in chapter two.

BRK

Top right of the numeric keypad. The function of the BRK is to BReaK into a program which is currently being RUN.

TAB

This key (the 4 of the numeric keypad) moves the cursor over the text, from left to right, in leaps of eight characters. This key proves very useful when editing large sections of code.

CURSOR CONTROL KEYS

UP ARROW

This key, the 5 on the numeric keypad, moves the cursor one line up the screen. However, this key will not work on the BASIC EDIT screen, so you must use the left and right cursor control keys to move around text on this particular screen.

LEFT ARROW

This key, the 1 on the numeric keypad, moves the cursor one space to the left each time it is pressed.

RIGHT ARROW

This key, the 3 key on the numeric keypad, moves the cursor one space to the right each time it is pressed.

DOWN ARROW

This key moves the cursor down one line, but, as with the up arrow key, it will not work on the BASIC EDIT screen.

HOME

This key, the 2 on the numeric keypad, returns the cursor to the beginning of the screen.

INS

This key is a toggle used when editing text on the screen. The INS key, when pressed for the first time, enables you to INSert text at the current cursor position. It should be noted that this facility is turned off each time the <RET> key is pressed.

DEL

This key DELetes the character under the cursor, and will continue to delete subsequent characters until it is released.

ENT/CLS

ENT/CLS, situated at the bottom right-hand corner of the numeric keypad, is a dual purpose key. It can either be used to CLear the Screen or, when SHIFTed, ENTer a line of text. When it is used to CLear the Screen it will only clear the BASIC EDIT screen (the bottom four lines of the screen).

The CLS key followed by <RET> can also be used to terminate a number of operations. For example, CLS-<RET> disables the AUTO line numbering facility, facilitates a return to BASIC from Noddy and Assembler and can also be used to delete a Noddy page.

FUNCTION KEYS

On the far right of the machine are 8 keys labelled F1 through to F8. These are known as function keys and will be discussed later in the text.

CLEARING THE SCREEN

Up until now everything you've typed in has been confined to the bottom of your screen. On power-up your micro only provides access to the BASIC EDIT screen, which utilises the bottom four lines of your VDU. However, this screen is invaluable since it enables us to enter programs and issue commands.

Type in some characters until you have familiarised yourself with the Memotech's keyboard. Once you feel quite at home with the layout you'll be ready to move on to bigger and better things. However, before we enter the wonderful world of BASIC, you must learn how to clear this screen of unwanted text. This can be achieved by either RESETting the machine (by simultaneously pressing the two keys on either side of the space bar), or by typing the CLS following by <RET> or by pressing F2 followed by <RET>.

The first of these two methods will clear the entire screen and erase any information currently stored in the computer's memory. The second sequence (CLS/ENT) simply clears the BASIC EDIT screen, but keeps the contents of the memory intact.

We have already established the importance of precision when issuing instructions to computers. In the next few chapters we'll introduce you to the foundations of the BASIC language, you should bear in mind the need for accuracy when typing in the example programs.

There are two different ways of issuing instructions in MTX BASIC. The most common method is in the form of a program, which is a numbered list of commands stored in the Memotech's memory. The second option is to enter the commands directly. This method executes commands the moment <RET> key is pressed but does not store the actual code in memory. Since direct commands force immediate response from the computer, we'll use this mode of instruction as a means of introducing you to the principles of BASIC.

We have already established that computers are very fussy about the way in which instructions are formatted. This need for precision extends to the spacing of commands. Nearly all MTX commands must be followed by a space if they are to be accepted as a legitimate entry. However, Memotech BASIC is easier to use than the majority of dialects since most keywords can be entered in an abbreviated form which automatically includes a following space.

GETTING INTO PRINT

The first BASIC keyword that we'll examine is the PRINT statement. This instruction, as you might expect, is used to PRINT items of data to the screen. The characters to be PRINTed when using the simplest PRINT format must always be enclosed within quotes. Type in:

PRINT "HI THERE EVERYONE I AM 4 YEARS OLD!"

and press the <RET> key to activate the command. As soon as the <RET> key is pressed - HI THERE EVERYONE I AM 4 YEARS OLD! - will be PRINTed to the screen on the line below the command. As soon as the computer encounters PRINT followed by characters within quotes, it stops thinking and mindlessly displays the quoted material to the screen. We'll see a little later there are PRINT formats which allow us to PRINT material without the use of quotes. However, for the moment note that the simplest form of the PRINT statement requires information to be enclosed between double quotes. Try the following example which excludes the quotes and see what happens.

PRINT MEMOTECH <RET>

This time, instead of 'MEMOTECH' being PRINTed to the screen our trusty micro has come up with one of its numerous error messages: 'Undefined'. For now do not concern yourself with the meaning of this message (it will be explained later in the chapter). Simply note that double quotes are required whenever you want the MTX to passively PRINT material to the screen. Erase the error message by pressing EOL or CLS followed by <RET>. We should probably make it clear that 'characters' is an all-embracing term used to describe the letters, numbers and symbols that can be displayed using a PRINT statement. For example:

PRINT "ABC 123 !#:;*" <RET>

will PRINT all the characters enclosed within the quotes to the screen. Like a 'gaggle' of geese, this collection of quoted characters is referred to as a 'string'.

By now you're probably very bored of typing PRINT each time you want to create a PRINT statement. However, as we mentioned earlier, your micro allows you to abbreviate keywords and the short form of PRINT is P. so, make life easier for yourself and whenever you come across a PRINT statement, just type in 'P.'. On pressing <RET> you'll notice that the statement will be displayed in its unabbreviated form.

PRINT AND NUMBERS

BASIC offers alternative PRINT formats which force the computer to consider PRINT material in a less literal manner than the quoted string displays we have just discussed. For example, we can use the Memotech as a rather expensive calculator by asking it to PRINT the calculation of numbers. In the light of what we have just discovered you shouldn't be surprised that:

PRINT "8-4" <RET>

displays 8-4 on the screen. In other words, the numbers contained within quotes will be literally reproduced. However, by omitting the quotes we can get the computer to perform the actual calculation. Thus:

PRINT 8-4 <RET>

will PRINT 4 to the screen. By omitting the quotes we force the computer to consider the numbers following the PRINT statement as a numerical expression, which must be calculated before it is PRINTed.

Let's try a few more PRINT statements and make sure that you're completely at home with this valuable command. (We will assume that you've recognised the need to press <RET> to activate a command, so we won't mention it again.)

PRINT 8+4-2

will PRINT 10 to the screen.

The asterisk (*) is the micro's equivalent of the multiplication sign and is used to avoid any confusion with the letter x. Thus:

PRINT 2*4

will PRINT 8 to the screen. However, if we ask the Memotech to calculate 3*3+4 we have to be clear about what we're asking the micro to work out for us. In other words, do we mean (3*3=9) plus 4 (9+4=13), or 3+4 (3+4=7) multiplied by 3 (7*3=21). To avoid this sort of confusion the computer assigns a set of priorities to each mathematical operator. The following table shows the priorities of these arithmetical operators in descending order:

()	Brackets
^	Exponential
* /	Multiplication and Division
+ -	Addition and Subtraction

So, the MTX will always deal with any expression within brackets as its first priority. Thus:

PRINT 6+(2*2)

will return 10, since this expression is equivalent to 6+4.

The next operational priority is exponentiation, the process of raising one value to the power of another (for example two squared which is 2*2). To refresh your memory, 2 raised to the power of 4 is 2*2*2*2=16. The exponential numerical operator uses the ^ symbol which you will find next to the equals sign on the QWERTY keyboard. Thus:

PRINT 2^4*3

will return 48. In other words, 2^4 (or 2*2*2*2=16) which is then multiplied by 3 (16*3=48).

The next operators on the priority list are multiplication and division which are assigned an equal priority by the computer. Thus:

PRINT 2+2*3

will return 8.Our micro will first calculate 2*3=6 (since multiplication has a higher priority than addition) and then add 2, returning 8. Similarly:

PRINT 2+6/2

will return 5. First 6/2=3 is calculated and then 2 is added to the result, returning 3+2=5. When operators with an equal priority are used in the same calculation, the computer reads the expression from left to right. Thus:

```
PRINT 2/3*6
```

will return 4.

Since addition and subtraction are also assigned equal priority, they are also interpreted from left to right when both are used in the same expression. Thus:

```
PRINT 2-4+6
```

will return 4 and:

```
PRINT 6*2-3
```

will return 9 because subtraction has a lower priority than multiplication. Let's try a more complex calculation which should clarify the need for a system of pre-defined priorities.

```
PRINT 2*4^2/(6-3)+4
```

Since brackets take the highest priority of all the numerical operators, the bracketed expression (6-3) is calculated first, thus changing our expression to 2*4^2/3+4. The next calculation involves exponentiation. Once 4 has been raised to the power of 2 our calculation will look like this: 2*16/3+4. As multiplication shares the next highest priority with division but on a left-to-right basis, our expression will be 32/3+4 after the computer has performed its multiplication. Penultimately it will work out 32/3 to return 10.6666667, and finally add 4 to produce 14.6666667.

We've seen how the inclusion or exclusion of quotes alters the manner in which the computer interprets a PRINT statement. We'll now move on to see how other forms of punctuation can be used to control screen displays. By using a semi colon (;) it is possible to join one sequence of PRINT items to the next. For example:

```
PRINT "3+2=";3+2
```

The effect of a semi colon is to force the cursor to remain at the PRINT position it reached after displaying the last print item. Thus, the next time the computer comes across a PRINT item, whether it appears in the same statement or in a subsequent PRINT statement it will tack it on to the end of the last item displayed. Therefore the above statement will PRINT out:

```
3+2= 5
```

You'll notice a space between the equals sign and the five. This space is referred to as a leading space which leaves room for a minus sign in the event of a negative number. Let's try PRINTing a negative number:

```
PRINT "3-4=";3-4
```

This statement will PRINT:

```
3-4=-1
```

As you can see, the leading space is now occupied by the minus sign. Key in the following statement:

```
PRINT 1;-6
```

and '1-6' is PRINTed to the screen. It is important to remember that, as this example demonstrates, numbers are not displayed with a trailing space. We actually wanted the computer to PRINT 1 -6, but with

no trailing space it simply joined up the numbers and produced a display that looks like a calculation as opposed to a pair of individual numbers. However, by entering either:

```
PRINT 1;"  ";-6
```

or

```
PRINT "1 -6"
```

we can achieve the desired effect. The semi colon is one of two PRINT separators available in MTX BASIC. While the semi colon joins PRINT items, a comma sends the cursor to the next print field on the screen. Key in the following example:

```
PRINT 3,4
```

Having displayed the 3 at the first PRINT position on the screen, the cursor then moves on 7 spaces before PRINTing the 4. The Memotech's screen has 5 PRINT fields across the width of the screen, and each PRINT field can display 8 characters. By entering the following statement we can discover the first PRINT position of each field:

```
PRINT 1,2,3,4,5
```

If we try adding a further item to this statement the computer will simply move on to the first PRINT field on the next screen line:

```
PRINT 1,2,3,4,5,6.
```

LET IT BE

We have considered the PRINT command in depth primarily because it is an essential, flexible yet straightforward BASIC instruction. However, PRINT statements also allow us to painlessly introduce the concept of a program. This is because we can demonstrate most PRINT formats without having to rely on any other BASIC word. However, like the words of any language, the power of most the BASIC keywords can only be fully demonstrated in the context of other keywords. Thus, it will be necessary for some of the examples which follow to make use of commands and structures that will not be explained until later in the manual. On these occasions you can either look up the new statement in the keywords section or else contain your curiosity until the keyword in question pops up in this brief tutorial. Earlier in this chapter we asked the computer to:

Print Memotech

In other words, we tried to get the computer to PRINT a string without quotes. You'll remember that the Memotech refused to accept this instruction and an 'Undefined' error message was returned. This error message is a burst of disgruntled shorthand which tells you that you have tried to PRINT an undefined 'variable' called MEMOTECH. You will have to excuse the jargon, but since the concept of variables is central to computing you'll have to grit your teeth and grapple with it! The easiest way to think of a variable is as a box to which you can assign a name and into which you can place a value. In order to define a variable we use the BASIC keyword LET. If we had typed in:

```
LET MEMOTECH=24
```

followed by:

```
PRINT MEMOTECH
```

24 would have been PRINTed to the screen. In the first statement (LET MEMOTECH=24) we assigned the value of 24 to a variable that we called MEMOTECH. Thus, when we entered PRINT MEMOTECH the micro duly PRINTed the value that had been assigned to the aforementioned variable.

It is always advisable to create a variable name that will remind you of its role in the program. For example, suppose you wanted to set up a variable to store somebody's age. To avoid any ambiguity we could simply call the variable AGE.

```
LET AGE=25
PRINT "I AM";AGE;" YEARS OLD"
```

Our first line assigns the value of 25 to the variable called AGE. The next statement PRINTs out the information between the quotes (I AM) and the semi colon which follows ensures that the value of the variable AGE (25) is PRINTed directly next to it. The second semi colon informs the computer to PRINT the string between the quotes (YEARS OLD).

As we have already discovered it is important to make it clear to the Memotech whether characters should be considered as string or numeric values. When using variables we establish this distinction by selecting one of two variable formats. The type of variables we have discussed to date are known as numeric variables. AGE is a numeric variable because it stores a value which the computer can use in arithmetic operations. We can also create string variables by assigning a variable name to a string of characters. When a string variable is created a slightly different format is required. The LET statement is still used but the variable name must be followed by a dollar sign ($) and the string that is being assigned to that variable is enclosed within quotes. Thus:

```
LET NAME$="SPENCER"
PRINT "MY NAME IS ";NAME$
```

will PRINT - MY NAME IS SPENCER - to the screen, since NAME$ stores the characters - SPENCER. However, if we omit the dollar sign or the quotation marks, thus:

```
LET A=HELLO
```

the cursor will return to the beginning of the line and the dreaded 'Undefined' error message will appear. As with all BASIC statements, the computer will only accept instructions which use the correct format. In this case, since there is no dollar sign after the variable name, the micro is quite entitled to expect numeric data, so, it's hardly surprising that illegal string data produces error message. Similarly, in the following example the Memotech expects to find string data enclosed within quotes, but instead encounters characters without quotes. In this case a 'Mismatch' error is generated. This report informs us that we have tried to assign inappropriate data to a particular type of variable - i.e. string data to a numeric variable or numeric data (or characters without quotes) to a string variable.

A variable name can be of any length, but cannot contain a space or any symbols other than roman characters. The following example shows how variables got their name. As you'll see, the value assigned to a variable can be changed at any time:

```
LET A$="I AM"
LET B$=" YEARS OLD."
LET AGE=40
PRINT A$;AGE;B$
```

This will PRINT variables to the screen in the order specified. Now let's change the value held by the variable AGE and PRINT the last line again:

```
LET AGE=25
PRINT A$;AGE;B$
```

This time the line PRINTed displays the new value of AGE.

By now it's hopefully clear that it would be much simpler to put this list of commands into a program. In this way we can avoid retyping each line whenever we want to make a change or, for that matter, when we want to PRINT the variables.

The difference between a program and the direct commands we've been using up until now is quite straightforward. When writing a program each statement is given a line number. Before a new line can be typed in, the <RET> key must be pressed. Unlike direct commands, line numbered statements are not entered into the computer for immediate processing by <RET>, but stored in the computer's memory.
NB Entering a program line clears all variables.

BASIC PROGRAMMING

Let's key in the last sequence of statements again, but this time we'll give each instruction a line number.

```
10 LET A$="I AM"
20 LET B$=" YEARS OLD"
30 LET AGE=40
40 PRINT A$;AGE;B$
```

To enable the computer to execute this list of commands you must type in RUN and press <RET>. The computer will deal with the instructions according to the order established by the line numbers, starting with the lowest and finishing with the highest. When you RUN our example the screen is cleared and the characters PRINTed to the top-left hand corner of the screen. Type in the following line and then RUN the program again.

```
35 PRINT : PRINT : PRINT : PRINT : PRINT
```

This time the same sequence of characters are PRINTed lower down the screen. By using empty PRINT statements we force the cursor down one line per statement. In effect we're PRINTing a blank line to the screen. You'll notice that colons (:) have been used instead of semi colons. The effect of a colon is to allow a single program line to contain more than one statement. In most circumstances this format has exactly the same effect as creating a new program line.

So far we have made the somewhat reckless assumption that you have made no typing errors when keying in your commands. What is more likely is that you've had to re-type a line each time you've made an error! This is clearly the time to take a look at your micro's editing facilities.

Before we can edit a program we need to display it on the screen. Type LIST and press <RET>. The LIST command, when used on its own, LISTs the entire program to the screen. However, LIST statements can take a variety of formats which enable us to LIST specified sections of code or a particular program line. It's well worth familiarising yourself with the various permutations of this simple command since you'll find it essential to the development of any program. Note that L. is the abbreviation of the LIST command. By typing in LIST 20 the Memotech LISTs the entire program from line 20 onwards, LIST 20,20 will LIST line 20 only and LIST 20,40 will LIST lines 20 to 40 (inclusive) to the screen.

The only problem with LISTing a program is that the majority of programs are larger than your TV screen. So, when you LIST a program the computer scrolls it up the screen until it reaches the final program line. However, by pressing the PAGE key (top left of the numeric keypad) you can pause the LISTing and

examine the section(s) you're interested in. To continue the LISTing simply press the PAGE key for a second time. For more about PAGE refer to the keyboard section at the beginning of this chapter.

Having introduced the concept of line numbers, it's worth mentioning that the Memotech boasts an AUTO line numbering facility. This is accessed by the BASIC keyword AUTO which takes the format 'AUTO 100,10' where 100 is the first line number you require and 10 is the step between line numbers. Thus, AUTO 100,10 will create lines 100, 110, 120, 130.... and so on until the CLS/ENT key (followed by <RET>) is pressed.

EDITING ON THE MTX

Having LISTed your program to the screen you can see that line 35 has been inserted between lines 30 and 40. Whenever you key in an extra line the computer will always insert it in the correct position. However, make sure that you don't use the same line number more than once, since the second statement will wipe out the original program line. You can use this to good effect when you want to delete a line, since by simply entering the appropriate line number and pressing <RET> you can delete the entire statement. Right, let's change our AGE variable back to 25. Type in:

EDIT 30

and line 30 will appear at the bottom of the screen with the cursor at the beginning of the line. Using the cursor control keys (the arrow keys on the numeric keypad) move the cursor along the line until it covers the 4 of 40. Press the DEL key (also on the numeric keypad) twice and the offending characters will be removed. Finally, type in 25 and press <RET>. LIST the program again and you will see that line 30 now reads:

```
30 LET AGE=25
```

To demonstrate the other editing facilities available on the Memotech let's change lines 10 and 20. Type in:

EDIT 10

and move the cursor along the line until it is over the A of AM. Press the DELete key twice (thus deleting AM) and then press the INSert key, which is to be found on the numeric keypad. The INSert key acts as a toggle which allows the INSertion of text. Whenever the <RET> key is pressed it is turned off, so you will always have to turn it back on again before you can INSert any text into a new line. Type in 'LIVE AT' and press <RET>. Your new line should look like this:

```
10 LET A$="I LIVE AT"
```

Now type in:

EDIT 20 or E.20

(E. is the abbreviation for EDIT) and move your cursor along until it is over the space inside the quotes. Press the INS key and press the space bar once, then press the INSert key once again (thus turning it off) and then type BLYTHE ROAD ". The ' YEARS OLD.' will now be replaced by BLYTHE ROAD producing a line that looks like:

```
20 LET B$=" BLYTHE ROAD"
```

The final touch our program requires is the inclusion of the programming equivalent of a name tag, which comes in the shape of a REM statement. REM is short for REMark or REMinder and the inclusion

of a REM has no effect on the way a program RUNs. Its sole function is to label a listing and REMind us what a program, or a section of code is meant to be doing. At first glance this might seem like a bit of a waste of time. However, it's actually very important to label your programs clearly with REM statements so that the code is easy to follow. Since our first program is fairly simple (and fairly dull!), a single REM is all that is required in this instance.

```
1 REM *** DEMONSTRATION PROGRAM ***
```

Since the MTX moves on to the next program line as soon as it encounters REM, you should never follow a REM statement with a colon and another statement. The second statement will never be processed!

The line numbers in the first version of this program were incremented in steps of ten. Although this step size is standard programming practice, you can increment your line numbers in any manner that takes your fancy without affecting the way it RUNs. However, an increment of ten gives us the ability to insert any extra lines we may require as the program develops.

INPUTTING DATA

Now that we've outlined the concept of a BASIC program, let's create a slightly more useful example and introduce you to a new keyword, INPUT. However, before you enter a new program you'll need to clear the computer's memory. You can do this by switching the MTX off and then on again, RESETting the machine (using the RESET buttons on either side of the space bar), or typing in NEW followed by <RET>.

The INPUT statement allows you to enter data while a program is RUNning. In simple terms, it can be seen as a cross between the PRINT and LET statements since it can be used to both display textual information and to define a variable. As soon as the computer encounters an INPUT statement it stops the program until the required data has been entered and <RET> pressed. Your entry is then stored in INPUT's variable.

If you haven't customised the statement by creating a textual prompt the computer will display a question mark to the screen when the INPUT statement is executed. The question mark simply indicates that the Memotech is awaiting information from the outside world. However, this doesn't actually give the user of a program much of an idea about the type of data expected. There are two ways around this problem. One alternative is to precede the INPUT statement with PRINT. For example:

```
5 REM  ****************
10 REM ***  INPUT  ***
15 REM ****************
20 PRINT "PLEASE ENTER YOUR NAME"
30 INPUT NAME$
```

The INPUT statement will still print the question mark prompt but at least the users will know what is expected of them. However, a more economical alternative is to include an explanatory prompt within the INPUT statement itself. Thus:

```
5 REM  ****************
10 REM *** INPUT 2 ***
15 REM ****************
20 INPUT "PLEASE ENTER YOUR NAME ";NAME$
```

NEW the memory and type in the following program. If nothing else it'll give you a chance to practice your newly acquired EDITing skills!

```
1 REM  **************
5 REM  *** INPUT 3 ***
10 REM **************
20 LET WHOLESALE=20
30 LET RETAIL=30
40 PRINT "HOW MANY ITEMS DID YOU SELL THIS WEEK?"
50 INPUT SOLD
60 LET PROFIT=(RETAIL-WHOLESALE)*SOLD
70 PRINT PROFIT
```

This program calculates and PRINTs the amount of profit produced in a given week by a suspiciously straightforward business. The INPUT statement (line 50) stops the program and the question mark prompt indicates that data is required. The program will only continue if the correct type of data is INPUT. In this case, since the INPUT variable SOLD is a numeric variable the Memotech will only accept a numeric INPUT. If you try entering string data the computer will reject your INPUT and follow the illegal entry with a second question mark.

The Memotech will not stop the program with an error report when the wrong type of data is entered. Instead, it allows you to make another entry and will continue to do so until the correct type of data is entered. It will respond in one of two ways. In our example we used INPUT without a string prompt so the program displayed yet another question mark on the following line. If we had used INPUT to display the quoted prompt, it would simply have re-printed the text on the next print line.

It is also possible to create multiple INPUTs by separating the INPUT statement's variables with a comma.

For example:

```
50 INPUT A,B$
```

(don't key this line in as it is only intended to demonstrate a multiple INPUT format.) However, when INPUT is used in this manner the user must enter data items in a single line separated by commas. For example:

12,fred

will assign the value 12 to the variable A and the string 'fred' to B$.

Delete line 50 by typing in 50 and pressing <RET>. Now change line 40 to read:

```
40 INPUT "HOW MANY ITEMS DID YOU SELL THIS WEEK? ";SOLD
```

Whilst we are polishing up our example let's make a couple of additional changes. In its present form our program simply PRINTs an unadorned, undefined result to the screen. Let's change line 70 so that it's slightly more informative.

```
70 PRINT "YOUR PROFIT FOR THIS WEEK IS";PROFIT;" POUNDS"
```

Try RUNning the program again and you'll find that this addition, although cosmetic, does make a difference. 'User friendly" is a cliche the world can well survive without, however, improvements like this help us to avoid the 'human hostile'!

GOTO

In a program painfully low on refinements, it might be a good idea to at least allow it to repeat itself so we can calculate the profit for more than one week. The statement we can use to achieve this is GOTO. GOTO is an extremely powerful command and should be treated with care since it disrupts the normal sequence of processing by line number by forcing the computer to GOTO a specified line number. If there are too many GOTOs in a program the code becomes impossible to follow and, more importantly, the final product is invariably inefficient. We'll consider the criteria of efficient program design a little later. For the moment we'll take the view that the use of GOTO is perfectly acceptable, so type in the following line and then RUN it once again.

```
80 GOTO 10
```

Since the computer will always return to line 10 once it has calculated the PROFIT, the only way you can exit this program is by pressing the BReaK key (top right of the numeric keypad). After RUNning this example for a while, the screen will fill up with information and generally become rather messy. Let's make another addition to our program which will clear the screen after each INPUT and demonstrate the value of the CLS command.

```
50 CLS
```

Try running the program again and you will find that as soon as your INPUT has been completed (i.e. as soon as you press the <RET> key) the screen will be cleared.

Since we have made quite a few modifications let's LIST the program before going any further:

```
1 REM  ***************
5 REM  *** INPUT 4  ***
10 REM ***************
20 LET WHOLESALE=20
30 LET RETAIL=30
40 INPUT "HOW MANY ITEMS DID YOU SELL THIS WEEK? ";SOLD
50 CLS
60 LET PROFIT=(RETAIL-WHOLESALE)*SOLD
70 PRINT "YOUR PROFIT FOR THIS WEEK IS";PROFIT;" POUNDS"
80 GOTO 10
```

DO YOU WANT ANOTHER GO?

Although our current version of the INPUT demo is much improved, the program would be far more professional if after each calculation we gave the user the option of either exiting the program or having another go. The INKEY$ command facilitates this extra touch of user participation.

When an INKEY$ statement is executed, the Memotech scans the keyboard to see if a key is being pressed and, if it is, it stores its value. The difference between this command and the INPUT statement is that INKEY$ in its 'raw' form does not actually stop a program. If a key is not being pressed at the precise moment the INKEY$ statement is being executed, the program simply moves on to the next program line. Another limitation of INKEY$ is that it only enables a single character string value to be stored. This said it does allow a value to be stored and the program to continue without the user having to press the <RET> key. However, we're not interested here in using INKEY$ to hold a value. Enter the following lines:

```
74 PRINT "WOULD YOU LIKE TO PERFORM ANOTHER CALC-ULATION? (Y/N)"
76 IF INKEY$="" THEN GOTO 76
80 IF INKEY$="N" THEN STOP ELSE GOTO 10
```

This addition gives the user the option of leaving the program or performing another calculation. Don't worry about the IF...THEN construction we've used in line 76 and the new line 80. This will be discussed later in the book. (If your curiosity is aroused have a look at the entry in the keyword section.) Line 76 scans the keyboard for a key press and if none is being made GOes TO the beginning of the line and repeats the check. When empty quotes are used in this manner they are referred to as a null string. In essence line 80 says 'if you pressed N then STOP the program or ELSE GOTO line 10'. In other words, unless you press the 'N' key the computer will repeat the process by GOing TO line 10. In this particular program we're obviously not interested in storing the value of the INKEY$ input. As long as it is not 'N' (for NO), we're simply interested in being able to INPUT a new SOLD variable. However, there are circumstances in which we'll want to use INKEY$ as a substitute for INPUT and this requires access to the value it stores. There are two ways in which this can be achieved. The first method is to consider INKEY$ as a string variable which can be used to store a single character string:

```
1 REM  ****************
5 REM  ***  INKEY$  ***
10 REM ****************
20 PRINT "ENTER ANY CHARACTER"
30 IF INKEY$="" THEN GOTO 30
40 PRINT INKEY$
```

Line 30 ensures that the program will not continue until a key is pressed and line 40 treats INKEY$ as a string variable that can be displayed on the screen. However, if we use an INKEY$ statement more than once in a program, it is only possible to store the most recent value assigned to INKEY$, any earlier values will be replaced by the latest key press. Thus it is usually necessary to store to value of the INKEY$ key to a string variable.

```
1 REM  ****************
5 REM  *** INKEY$ 1 ***
10 REM ****************
20 PRINT "ENTER ANY CHARACTER"
30 LET A$=INKEY$
40 IF A$="" THEN GOTO 30
50 PRINT A$
```

SUBROUTINES

We have seen how the GOTO command enables us to interrupt linear processing and direct control to a specified line number. We're now going to introduce one of the most important command structures available to the BASIC programmer which, although ostensibly an enhancement of the GOTO facility, actually holds the key to efficient coding in the Memotech's resident language. GOSUB resembles GOTO in that it also forces the computer to execute instructions from a line number defined by the statement rather than executing the line numbers in sequence.

However , appearances can be deceptive! GOSUB is a considerably more sophisticated command than GOTO because although it directs control to a specified part of the program, it also remembers the line number in which the GOSUB statement re-directed control. This means that once the instructions to which control has been directed by the GOSUB statement have been processed, a RETURN statement enables the computer to return to the program line following the original GOSUB. The section of code to which GOSUB statements direct control are known as subroutines. These 'mini-programs' are often responsible for operations which are required more than once in a program. By using GOSUB statements we can avoid replicating sections of code by simply calling the appropriate subroutines as and when required.

Unlike GOTO, the use of GOSUB statements is not frowned upon by programming purists. This is primarily because they store the RETURN address and thus promote the creation of programs comprising a series of clearly differentiated operations controlled by the main program. Such code is easy to follow and forces the programmer to plan the structure of a program before touching the computer. Subroutines are normally placed at the end of a program so that each discrete operation is clearly differentiated, and the processing sequence is immediately apparent from the GOSUB calls in the main program.

To demonstrate this structure we'll re-write our program incorporating the GOSUB...RETURN structure.

```
10 REM**********
20 REM***GOSUB***
30 REM**********
40 GOSUB 100
50 GOSUB 200
60 IF F=1 THEN  PRINT "GOODBYE": STOP
70 GOSUB 300
80 GOSUB 400
90 GOTO 40
95 REM
100 REM***VARIABLES***
110 LET WHOLESALE=20
120 LET RETAIL=30
130 LET F=0
140 RETURN
150 REM
200 REM*START/AGAIN?*
210 PRINT "WEEK'S PROFIT CALCULATION(Y/N) ?"
```

```
220 IF INKEY$="" THEN  GOTO 220
230 IF INKEY$="N" THEN  LET F=1
240 RETURN
250 REM
300 REM***INPUT/CALC***
310 INPUT "HOW MANY ITEMS DID YOU SELL THIS WEEK?";SOLD
320 LET PROFIT=(RETAIL-WHOLESALE)*SOLD
330 RETURN
350 REM
400 REM***RESULT***
410 CLS
420 PRINT "YOUR PROFIT THIS WEEK IS ";PROFIT;" POUNDS"
430 RETURN
440 REM
```

This program contains a number of new commands. For example line 200 starts a subroutine which determines whether you want to re-RUN the program. If you type in N the process will STOP and if you type in Y (or anything else for that matter) the INKEY$ command looks at your INPUT and moves to the following line, which RETURNs the control back to the main program. For further clarification of GOSUB refer to the keyword section.

The STOP command, as you might expect, halts the execution of the program, returning the 'Ready' prompt to the screen. With the use of a PAUSE statement we can PAUSE a program for a specified period of time. PAUSE must be followed by a number between 0 and 65535, the larger the number the longer the wait. Insert the following lines into our program.

```
85 GOSUB 500
500 REM***PAUSE***
510 PRINT : PRINT
520 PAUSE 3500
530 RETURN
```

and our final version of the program should look like this:

```
10 REM**********
20 REM***GOSUB***
30 REM**********
40 GOSUB 100
50 GOSUB 200
60 IF F=1 THEN  PRINT "GOODBYE": STOP
70 GOSUB 300
80 GOSUB 400
85 GOSUB 500
```

```
90 GOTO 40
95 REM
100 REM***VARIABLES***
110 LET WHOLESALE=20
120 LET RETAIL=30
130 LET F=0
140 RETURN
150 REM
200 REM*START/AGAIN?*
210 PRINT "WEEK'S PROFIT CALCULATION(Y/N)?"
220 IF INKEY$="" THEN  GOTO 220
230 IF INKEY$="N" THEN  LET F=1
240 RETURN
250 REM
300 REM***INPUT/CALC***
310 INPUT "HOW MANY ITEMS DID YOU SELL THIS WEEK?";SOLD
320 LET PROFIT=(RETAIL-WHOLESALE)*SOLD
330 RETURN
350 REM
400 REM***RESULT***
410 CLS
420 PRINT "YOUR PROFIT THIS WEEK IS ";PROFIT;" POUNDS"
430 RETURN
440 REM
500 REM***PAUSE***
510 PRINT : PRINT
520 PAUSE 3500
530 RETURN
```

SPAGHETTI PROGRAMMING

Earlier in this chapter we mentioned that excessive use of the GOTO statement causes confusion and inefficiency, as well as making a program difficult to follow. The example below demonstrates such abuse and definitely satisfies all the criteria of a 'spaghetti program'.

```
 1 REM  ******************************
 5 REM  *** HOW NOT TO USE GOTO - ***
 10 REM ******************************
 15 REM *** SPAGHETTI PROGRAMMING ***
 20 REM ******************************
 30 INPUT "WHAT IS YOUR NAME? ";NAME$
```

```
40 GOTO 70
50 PRINT "HELLO ";NAME$
60 GOTO 90
70 INPUT "HOW OLD ARE YOU? ";AGE
80 GOTO 50
90 PRINT "I WOULD NEVER HAVE GUESSED YOU WERE";AGE;" YEARS OLD"
```

You'll doubtlessly be horrified to discover that this mess RUNs in spite of itself. Hopefully it clarifies our contention that an excessive use of GOTOs makes life more than a little confusing!

Having introduced a number of important BASIC keywords, let's take a break and take a look at the MTX's function keys which allow efficient access to such commands. These keys are situated on the far right of the keyboard and have some BASIC keywords built into them. Try pressing F1. On hitting <RET> the keyword REM is displayed to the screen. Now type in the following line (SHIFT F1 means that you must press SHIFT and F1 simultaneously).

10 F1 F2 F3 F4 F5 F6 F7 F8 SHIFT F1 SHIFT F2 SHIFT F3 SHIFT F4 SHIFT F5 SHIFT F6 SHIFT F7 SHIFT F8

On pressing <RET> all the BASIC keywords stored in the function keys are displayed. The line displayed should look like this:

```
10 REM CLS   ASSEM   AUTO   BAUD   VS   CONT   USER   CRV
S   CLEAR   CLOCK   ATTR   COLOUR   INK   CSR   DATA
```

You won't recognise the majority of these commands since we've only encountered REM and CLS so far. However, it is an idea to bear this facility in mind since it's a great help to be able to enter a keyword by a single keypress.

Before we move on to the next chapter let's run through the essential points you should remember about each command we have looked at so far.

AUTO

1. Used to access the AUTO line numbering facility.
2. It takes the format AUTO x,y, where x is the start line number and y is the increment value.
3. Takes the abbreviation AU.

CLS

1. The abbreviation for the CLS command is 'C.'.
2. When entered via the CLS/ENT key it will only clear the EDIT screen.

EDIT

1. The abbreviation for this command is 'E.'.

GOSUB

1. A command which enables you to interrupt the linear execution of a program by directing control to a specified line number.

2. The section of code accessed by the GOSUB command is called a subroutine.
3. Stores return address and returns control to the line following the GOSUB statement when a subroutine is terminated by RETURN.
4. The intelligent use of subroutines is the key to well structured programs.
5. GOSUB takes the abbreviation GOS.
6. RETURN takes the abbreviation RET.

GOTO

1. A command that enables you to interrupt the linear flow of a program by directing control to a specified line number.
2. It should be used with care in order to avoid poorly structured programs.
3. Takes the abbreviation G.

INKEY$

1. Similar to INPUT but does NOT halt a program
2. No need to press <RET> to enter
3. Stores only one character
4. Can be made to equal a variable name, and can then be treated as a variable.
5. Takes the abbreviation INKE.

INPUT

1. Multiple INPUTs are available with commas.
2. The variable must be of the right type. i.e. if a numeric INPUT is required a numeric variable must be used in the INPUT statement.
3. The MTX displays a question mark prompt unless the INPUT statement uses a customised textual prompt in quotes. (For example, INPUT "What is your name?"; NAME$ would not display a question mark prompt, whereas, INPUT NAME$ would.)
4. The abbreviation for this command is INP.

LET

1. A variable name defined in a let statement can be of any length.
2. The variable name must only use letters from the standard Roman character set: it must include no spaces or punctuation and cannot be a reserved word (BASIC keyword).
3. When assigning a string variable the variable name must be followed by a dollar sign ($) and the string must be enclosed between quotes.
4. When assigning a numeric variable the value to be assigned must be either a number or a numeric expression.
5. Many dialects of BASIC permit the word LET to be an optional part of a LET statement. MTX BASIC requires its inclusion.
6. The abbreviation for this command is 'LE.'.

LIST

1. When used on its own the entire program is LISTed to the screen.
2. When the statement takes the syntax LIST 50 the computer LISTs the program from line 50 onwards.
3. When the statement takes the syntax LIST 50,50 the computer LISTs line 50 only.
4. When the statement takes the syntax LIST 50,80, the computer LIST lines 50 to 80.
5. The abbreviation for this command is L.

PRINT

1. Any information to be PRINTed to the screen must be enclosed within quotes.
2. Each new PRINT statement causes the information to be PRINTed on the following PRINT line.
3. A semi colon leaves the cursor at the current cursor position thus causing subsequent PRINT items to be PRINTed on the same PRINT line and immediately after any preceding items.
4. A comma moves the cursor to the next PRINT field causing any items which follow to be PRINTed at this position.
5. Quotes should be ommitted when PRINTing numbers, calculations or variables.
6. The abbreviation for the PRINT statement is P.

REM

1. Short for REMark or REMinder
2. Used to label programs or identify sections of code.
3. Takes the abbreviation R.

CHAPTER 3 : DEALING WITH DATA

Approaching the problem

In the previous chapter we demonstrated that GOTO, when misused, can create spaghetti programs which are both inefficient and invariably incomprehensible. BASIC purists tend to frown on the use of GOTO statements because they offer the inexperienced programmer a ready means of 'patching' poorly conceived code. To a certain extent, any collection of BASIC statements can be cobbled together into something which can be persuaded to RUN. However, if you don't systematically plan your approach to a problem it's more than likely that you'll end up patching inoperative code with GOTO statements.

In order to avoid such a situation it's well worth taking time out to determine the most efficient way of writing the program. This isn't to say that programs should be drafted in long hand and then keyed-in. All such an approach involves is the breaking down of a problem into clearly defined sections (or modules) of BASIC instructions. Once you've dismantled a problem into manageable sub-problems it's relatively simple to determine the order in which the appropriate modules of BASIC code should be processed to produce an efficient program. Let's take a simple problem and then run through the factors that should be considered before the program itself is actually written.

The Problem

Assume that we want to work out the area and circumference of a circle whose radius is to be given by the user of the program. In addition, we also require the values returned by the program to be corrected to two decimal places.

Our first step should be to determine how many self-contained operations are required in the solution of this problem. This approach will result in the creation of a 'task' list:

1. Display the function of the program.
2. Provide an INPUT facility, so that the user can enter the radius on which subsequent calculations will be based.
3. We need to assign to a variable the formula for calculating the circumference of a circle.
4. We need to assign to a variable the formula which calculates the area of a circle.
5. We need a routine that will correct results of the calculations to two decimal places.
6. We need a method of displaying results to the screen.

By comparing the above list of tasks with our original brief, we can be sure that we have identified every aspect of our problem.

What is clear from our task list is that one of the processes must be performed twice for each radius entered. Although the calculations to determine the area and circumference of a circle will obviously be different, the results of both calculations need to be rounded to two decimal places. Thus the rounding procedure can obviously be coded as a subroutine to avoid having to write the same set of instructions for the result of each calculation.

Having decided that process number five can be used twice, we're immediately faced with another problem. The results of our two calculations are held in two different variables, and both variables must be applied to the same process.

In other words, if the variable A is assigned the value of the area of a circle, and C holds the value of the circumference of a circle, we have a problem. How can we use two different variables in the same routine to round to two decimal places? To make this problem a little clearer we must jump ahead and take a look at the subroutine we will have to create to round to two decimal places.

This routine contains a new keyword which will be explained later in the chapter. For the moment, don't worry about the calculation involved, but concentrate on the problem of using two variables in the same

calculation. The new keyword in question is the INT function. The keyword section contains a full description of the INT statement and, if you're interested, you should take a look at this entry before reading any further. However, all you really need to know is that INT rounds a decimal number down to the next lowest whole number.

Let's construct our rounding subroutine so that it rounds the result of calculating the area of a circle to two decimal places. Remember the variable A=Area of the circle.

```
200 REM ****************************
203 REM *** ROUNDING SUBROUTINE ***
205 REM ****************************
210 LET A=INT(100*(A+.005))
220 LET A=A/100
230 RETURN
```

This will round our circle area (A) to two decimal places. The problem with variables should now be clear. Whilst the result of the Area calculation can now be rounded, we cannot use this routine to round the circumference result (because this result is represented by the variable C not A). The only way around this problem is to avoid using either C or A in the rounding routine. Let's say that Z is any value to be rounded to two decimal places. Thus we can add two more processes to our list:

7. Convert Area variable into rounding variable
8. Convert Circumference variable into rounding variable

Now that our list of tasks is ostensibly complete, we can turn to the keyboard and start coding the program. Task one simply PRINTs the programs objective to the screen:

```
1 REM  ****************************
3 REM  * STRUCTURED  PROGRAMMING  *
5 REM  ****************************
7 REM  ***     DISPLAY INTENT    ***
9 REM  ****************************
10 CLS
20 PRINT "      THIS PROGRAM CALCULATES THE "
30 PRINT "CIRCUMFERENCE AND AREA OF ANY CIRCLE ":
PRINT
```

This module of code is quite clear. Lines 20 and 30 display the options offered by the program, and line 10 is a statement to clear the screen. Now we need to allow the user to INPUT the radius value to be used in subsequent calculations:

```
40 REM ****************************
43 REM ***      USER INPUT        ***
45 REM ****************************
50 PRINT "PLEASE ENTER THE RADIUS OF THE CIRCLE.":
 INPUT R
```

As soon as the user enters a value we can perform our first calculation; the circumference of a circle with radius R. The formula for this is 2*PI*R. So our next lines will read:

```
60 REM ****************************
63 REM ***  CIRCUMFERENCE CALC  ***
65 REM ****************************
70 LET C=2*PI*R
```

Since we are going to require the result of this calculation to be processed by our rounding subroutine, we must change the name of the circumference variable C to Z:

```
80 REM ****************************
83 REM ***  VARIABLE CONVERSION ***
85 REM ****************************
90 LET Z=C
```

Next we need to call the rounding subroutine and, once the rounding process has been completed, PRINT the result to the screen:

```
93 REM ****************************
95 REM ** CALL  ROUNDING ROUTINE **
97 REM ***  AND  PRINT  RESULT  ***
99 REM ****************************
100 GOSUB 200
110 PRINT : PRINT "THE CIRCUMFERENCE OF A CIRCLE W
ITH A": PRINT "RADIUS OF";R;" IS";Z
```

Now we need to assign the expression which will calculate the Area of a circle to a variable. The formula for this calculation is PI*R ^ 2, so our next line must read:

```
120 REM ****************************
123 REM ***   AREA CALCULATION   ***
125 REM ****************************
130 LET A=PI*R^2
```

The variable A must now be made equal to the rounding variable Z:

```
140 REM ****************************
143 REM ***  VARIABLE CONVERSION ***
145 REM ****************************
150 LET Z=A
```

Once again we must call the rounding subroutine and then display the result to the screen:

```
160 REM ******************************
163 REM * ROUND AND DISPLAY RESULT *
165 REM ******************************
170 GOSUB 200
180 PRINT : PRINT "THE AREA OF A CIRCLE WITH A RAD
IUS": PRINT "OF";R;" IS";Z
```

We have now completed all the processing modules of our program. Lines 10-180 provide the means of INPUTing data, assigning variables and displaying results. The subroutine in line 200 performs the rounding operation for both calculations. We must now include a closing statement to prevent the control module from running into the subroutine:

```
190 STOP
```

When we initially wrote the subroutine it was coded to deal with the result of the Area calculation. Since we have created the rounding variable Z we must now make a couple of minor changes to the original subroutine:

```
200 REM ******************************
203 REM *** ROUNDING SUBROUTINE ***
205 REM ******************************
210 LET Z=INT(100*(Z+.005))
220 LET Z=Z/100
230 RETURN
240 REM ******************************
243 REM ***  END OF SUBROUTINE   ***
245 REM ******************************
```

Our complete program now looks like this:

```
1 REM  ******************************
3 REM  * STRUCTURED  PROGRAMMING  *
5 REM  ******************************
7 REM  ***     DISPLAY INTENT   ***
9 REM  ******************************
10 CLS
20 PRINT "      THIS PROGRAM CALCULATES THE "
30 PRINT "CIRCUMFERENCE AND AREA OF ANY CIRCLE ":
PRINT
40 REM ******************************
```

```
43 REM ***      USER INPUT        ***
45 REM ******************************
50 PRINT "PLEASE ENTER THE RADIUS OF THE CIRCLE.":
 INPUT R
60 REM ******************************
63 REM ***  CIRCUMFERENCE CALC  ***
65 REM ******************************
70 LET C=2*PI*R
80 REM ******************************
83 REM ***  VARIABLE CONVERSION ***
85 REM ******************************
90 LET Z=C
93 REM ******************************
95 REM ** CALL  ROUNDING ROUTINE **
97 REM ***  AND  PRINT  RESULT  ***
99 REM ******************************
100 GOSUB 200
110 PRINT : PRINT "THE CIRCUMFERENCE OF A CIRCLE W
ITH A": PRINT "RADIUS OF";R;" IS";Z
120 REM ******************************
123 REM ***   AREA CALCULATION   ***
125 REM ******************************
130 LET A=PI*R^2
140 REM ******************************
143 REM ***  VARIABLE CONVERSION ***
145 REM ******************************
150 LET Z=A
160 REM ******************************
163 REM * ROUND AND DISPLAY RESULT *
165 REM ******************************
170 GOSUB 200
180 PRINT : PRINT "THE AREA OF A CIRCLE WITH A RAD
IUS": PRINT "OF";R;" IS";Z
190 STOP
200 REM ******************************
203 REM *** ROUNDING SUBROUTINE ***
205 REM ******************************
210 LET Z=INT(100*(Z+.005))
220 LET Z=Z/100
230 RETURN
```

```
240 REM ******************************
243 REM ***  END OF SUBROUTINE   ***
245 REM ******************************
```

If you compare the final listing with our original itemisation of tasks, you'll see that each list item is represented by an appropriate module of BASIC code. Ideally you should always tackle a programming problem in this way. When a program's objectives are as straightforward as our example's such an approach is almost inevitable since, apart from the variable problem, the coding is so simple that the code almost writes itself. When a problem is more complex, it is considerably more difficult to hammer out a task list that relates so literally to the finished BASIC program.

The main reason for this is that when problems become more complex each process in the initial task list could easily be as involved as the complete program we have just written. Under these circumstances it's easy to overlook a complication like the need for variable conversion. On the other hand, as long as the major processing requirements have been correctly identified, a slight adjustment to a single module hardly constitutes a disaster.

The moral of all this is quite simple. The biggest mistake any programmer can make is to start churning out instructions which attempt to solve the entire problem in one fell swoop. The art of writing efficient and readable programs is primarily determined by the ability of the programmer to break down a problem into a series of smaller problems. As you have probably realised, individual BASIC commands can only be applied to very specific problems. The flexibility of the language as a whole is only apparent when the commands are creatively combined within a program. This said, it's precisely the specificity of each command which necessitates the dismantling of a problem into manageable sub-problems before it is coded. In this way individual tasks can be matched with BASIC modules on a one to one basis.

NUMBER SYSTEMS GALORE

If we look over the task list for our last program, we can identify three essential elements common to any program. In order to solve any problem your computer requires data to process, instructions which determine how this data is processed and a method of displaying or communicating the results of the processing to the user. Before we can go on to look at the sophisticated sound and graphics potential of your MTX we must be clear about the kind of data it will accept, and the way in which such data can be processed.

Our profit program in the previous chapter initially set up data in the form of variables. It then defined a variable which calculated profit and finally PRINTed the result to the screen. This program dealt solely with decimal numeric data (as opposed to both string and numeric data). There are, however, many different types of number systems, and one important method of presenting numeric data on micros is in the form of exponential numbers.

The exponential number system is used to express very large and very small positive numbers. If a value exceeds 999,999,999 (nine characters long) the MTX will present it in exponential notation which takes the format of 1.02E+09 or 1.02E-09. When an exponential value is expressed in the positive format, you must move the decimal point to the right to obtain the value in its normal format (in this case nine times, giving the number 1,020,000,000). If the exponential number is expressed in the negative format, you must move the decimal point to the left (in this case nine times giving the number 0.00000000102). The largest decimal representation that the MTX can hold is approximately 1.7014118E+38 and the smallest is approximately 9.9999999E-39.

In essence, the Memotech can deal with numbers between 999,999,999 and -999,999,999 in the normal way, but if a number is greater or smaller it will be returned in the exponential format that we have just outlined.

Another number system that the MTX (and all micros) deal with is the binary system. Binary numbers use a sequence of zeroes and ones to represent a value and it is this system that micros use to store data in memory. The system is appropriate because the electronic switches at the heart of all computers can only be either on or off, and a binary 1 represents a switch being on and the 0 represents off.

The number system with which most of us are familiar is decimal, or base 10. The binary system is in base 2. Let's do a bit of counting in binary and then see how it operates.

DECIMAL		BINARY
1	=	1
2	=	10
3	=	11
4	=	100
5	=	101
6	=	110
7	=	111
8	=	1000
9	=	1001
10	=	1010
11	=	1011
12	=	1100
13	=	1101
14	=	1110
15	=	1111

Before we take a look at the binary system we'll quickly run through the basic principles of the decimal system and then make some comparisons. When we have a number like 1987, one thousand, nine hundred and eighty-seven, we are actually saying:

1 x 10^3	=	1000
9 x 10^2	=	900
8 x 10^1	=	80
7 x 10^0	=	7
		1987

On this principle, the binary number 11011 can be broken down thus:

BINARY			DECIMAL		
1	1 X 2^4	= 10000	2^4*1	=	16
1	1 X 2^3	= 1000	2^3*1	=	8
0	0 X 2^2	= 0	2^2*0	=	0
1	1 X 2^1	= 10	2^1*1	=	2
1	1 X 2^0	= 1	2^0*1	=	1
		11011			27

The last number system that we'll consider in this section is the hexadecimal (or hex) system. This system is base 16. When counting using the hexadecimal number system the letters A-F are used as well as numbers, thus:

DECIMAL	HEXADECIMAL
1	1
2	2
3	3
4	4
5	5
6	6

7	
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13
20	14

By using the same system we used for decimal and binary numbers, we can break down the hex number 53 thus:

HEX		DECIMAL
5	5 X 16^1 = 50	16^1*5 = 80
3	3 X 16^0 = 3	16^0*3 = 3
	53	83

These are the only three number systems you'll have to cope with when using the MTX. If you find them difficult to digest en masse don't fret! At this point it's enough that you recognise in addition to the decimal system, micros utilise a number of other value representation systems.

ASCII CODES

Characters and symbols are not stored in your computer's memory in the form in which they are represented on the screen. Instead, each character is represented by a number and such numbers are called ASCII codes. ASCII is pronounced as-key and is an acronym for American Standard Code for Information Interchange. You will find the complete list of the ASCII codes in Appendix 4.

By using the BASIC keyword ASC we can find out the ASCII code of any given character. The following program PRINTs out the ASCII code of your INPUT.

```
1 REM ************
3 REM ***  ASC  ***
5 REM ************
10 INPUT "PLEASE ENTER A SINGLE CHARACTER ";A$
20 PRINT "THE ASCII CODE FOR ";A$;" IS ";ASC(A$)
30 GOTO 10
```

To exit this program you will have to press the BRK key. You can only use the ASC command to determine the numerical representation of strings. These can either be presented in the variable format above (ASC(A$)) or in the literal format:

ASC("7")

By PRINTing the latter format the ASCII code for the number seven will be returned (55).

This program could have been written just as effectively using INKEY$. Thus:

```
1 REM ************************
3 REM ***  ASC AND INKEY$  ***
5 REM ************************
10 PRINT "PLEASE ENTER A SINGLE CHARACTER "
20 LET A$=INKEY$
30 IF A$="" THEN  GOTO 20 ELSE  PRINT "THE ASCII C
ODE FOR ";A$;" IS ";ASC(A$)
40 PAUSE 3500 : CLS
50 GOTO 10
60 REM ******************************
63 REM * PRESS BRK TO EXIT PROGRAM *
65 REM ******************************
```

Input is stored in the variable A$. If a key is not being pressed the program GOes TO line 20: otherwise it PRINTs out the ASCII code of the key pressed.

CHR$ is a command that works in the opposite direction to ASC. It turns ASCII codes into the characters they represent (as opposed to converting characters to their ASCII codes). The number following the CHR$ command must be between 0 and 255 and contained in brackets. This said, the following codes cannot be PRINTed to the screen since they return an escape error message: 1, 2, 6, 14, 15, 16, 17, 18, 19, 27. Only codes 33-126 actually PRINT characters to the screen. However, the remaining codes are useful because they allow you to use CHR$ in conjunction with PRINT to access other display functions such as cursor control. Key in the following program and try experimenting with this command.

```
10 REM **************
13 REM ***  CHR$  ***
15 REM **************
20 INPUT "PLEASE ENTER A NUMBER (33-126) ";A
30 PRINT A;" IS THE ASCII CODE FOR ";CHR$(A)
40 GOTO 10
```

Let's try a few more interesting examples. For example, if you enter CHR$(4) the screen turns a lilac colour, while CHR$(24) turns it black. CHR$(6) will turn your display lilac, and CHR$(12) will have the same effect as CLS.

STRINGS TO NUMBERS

In the previous chapter we saw how the MTX examines the contents of a string variable (or quoted string) in very literal terms. It will not perform calculations on numeric representations defined as strings. Fortunately, the Memotech is armed with the VAL command which enables us to convert such a string into its numeric value. Key-in the following program and we will then run through it and see how it works:

```
1 REM **************
3 REM ***  VAL   ***
10 INPUT "PLEASE ENTER A NUMBER ";A$
20 LET A=VAL(A$)
25 PRINT : PRINT : PRINT
30 PRINT "NOW THAT YOUR INPUT HAS BEEN CONVERTED"
40 PRINT "FROM A STRING VARIABLE TO A NUMERIC "
50 PRINT "VARIABLE WE CAN PERFORM THE FOLLOWING "
60 PRINT "CALCULATION"
70 PRINT : PRINT : PRINT
80 PRINT "4 *";A;" + 1 =";4*A+1
```

Your INPUT is stored in the string variable A$, the VAL statement then converts A$ into the numeric variable A in line 20 by assigning the VALue of A$ to the numeric variable A. This conversion enables us to perform the calculation in line 80. If the first character of the VAL's argument is not a plus (+) or minus (–) sign, a space, decimal point or a number the Function will return a zero. If there are any non-numeric characters in the argument the Memotech will ignore the string from that point on. Thus the string 1.5h9 will be considered to be 1.5 and the 9 will be ignored.

AND BACK AGAIN

The complementary function to VAL is STR$. In the same way VAL turns strings to numbers, STR$ turns numbers into strings. It's action is quite straightforward, as you'll see if you RUN the following example which demonstrates the use of STR$ in conjunction with VAL. For a full explanation of these functions turn to the appropriate entry in the keywords section.

```
1 REM **************
3 REM ***  STR$  ***
5 REM **************
10 LET A$="1.52"
20 PRINT STR$(VAL(A$)*2/3.6)
```

STRING FUNCTIONS

As our next example program demonstrates, if you're attempting to format screen displays from user INPUTs it's often important to know the length of a particular string. This operation can be performed by the string function LEN, which must be followed by a space before its first bracket to avoid an error message.

```
1 REM  **************
5 REM  ***   LEN  ***
10 REM **************
20 INPUT "PLEASE ENTER A THREE DIGIT NUMBER ";A$
30 LET L=LEN (A$)
40 IF L<>3 THEN  GOTO 20 ELSE  PRINT A$,A$,A$,A$,A
$,A$
```

LEN can only be used with string variables, so to define the length of numeric variables you must make use of the STR$ function. However, it is important to note that when you convert a number into a string the leading space is incorporated into the string.

```
1 REM  **************
5 REM  *** LEN 2  ***
10 REM **************
20 INPUT "PLEASE ENTER A THREE DIGIT NUMBER ";A
30 LET A$=STR$(A)
40 LET L=LEN (A$)
50 IF L<>4 THEN  GOTO 20 ELSE  PRINT A$,A$,A$,A$
```

You will notice that we've allowed for the leading space by restricting the LENgth of the INPUT to four characters.

MTX BASIC offers a variety of commands which facilitate the manipulation of strings. Strings can be sliced, generally mutilated, and then painlessly joined together again. In the next few pages we'll take a look at the functions which facilitate such savagery and the circumstances in which they can be usefully employed.

STRING MANIPULATION

Having created and assigned a string to a variable, it's possible to link it to other string variables by using a process called concatenation. This rather intimidating term simply means that the computer permits string addition, enabling strings to be joined together using the addition sign (+). Key in the following example to see this in action.

```
10 REM ************************
13 REM ***  CONCATENATION  ***
15 REM ************************
20 INPUT "WHAT IS YOUR  NAME? ";NAME$
30 LET HI$="HELLO"
40 LET QUERY$="HOW ARE YOU?"
50 LET SPACE$=" "
60 PRINT HI$+SPACE$+QUERY$
70 INPUT ANSWER$
80 LET RESPONSE$="IS FEELING"
90 PRINT NAME$+SPACE$+RESPONSE$+SPACE$+ANSWER$
```

SLICING STRINGS

Using either LEFT$, RIGHT$ or MID$ we can extract a specified character or sequence of characters from a string. The following program demonstrates how LEFT$ accomplishes a 'string slice':

```
10 REM ***************
13 REM ***  LEFT$  ***
15 REM ***************
20 LET A$="WINE"
30 LET B$="DOWN"
40 LET C$=LEFT$(A$,3)
50 LET D$=LEFT$(B$,3)
60 PRINT C$+D$
```

This example extracts a specified number of 'left-most' characters from A$ and B$. The new strings created by the operation are assigned to the variables C$ and D$. The number of characters to be extracted are specified in lines 40 and 50, where the function's first parameter is the string to be sliced and the second the number of characters to be extracted. Thus LEFT$(A$,2) will extract the two left-most characters from A$. It is not necessary to assign LEFT$ to a variable. We could have written the program like this:

```
10 REM ***************
13 REM *** LEFT$ 2 ***
15 REM ***************
20 LET A$="WINE"
30 LET B$="DOWN"
40 PRINT LEFT$(A$,3)+LEFT$(B$,3)
```

RIGHT$ works in much the same way as LEFT$, except that the numeric parameter (let's call it 4) extracts the four right-most characters from the specified string. Thus:

```
10 REM ***************
13 REM *** RIGHT$  ***
15 REM ***************
20 LET A$="AREA"
30 LET B$="DEAL"
40 PRINT RIGHT$(A$,3)+RIGHT$(B$,1)
```

This program extracts the three right-most characters from A$ and the right-most character from B$. The extricated characters are then concatenated to produce REAL, which is PRINTed to the screen.

The last of the Memotech's string slicing functions is MID$ which, although similar to LEFT$ and RIGHT$, is slightly more sophisticated. MID$ uses three parameters:

MID$(A$,X,Y)

The first (A$) is the string from which the characters are extracted, the second (X) specifies the first character to be extracted and the third (Y) determines the number of characters to be extracted. So if we want to extract the string 'NIGH' from the string 'MIDNIGHT' our program will look like this:

```
10 REM ***************
13 REM ***  MID$   ***
15 REM ***************
20 LET A$="MIDNIGHT"
30 PRINT MID$(A$,4,4)
```

This program forces the computer to extract four characters from A$ starting with the fourth character. If the second parameter (Y) exceeds the length of the string, the statement will return the entire string starting with X. So, by changing line 30 of our example to read:

```
30 PRINT MID$(A$,4,20)
```

the sub-string NIGHT will be returned.

NUMBER FUNCTIONS

Having run through the string functions available in MTX BASIC, let's turn our attention to the numeric functions.

ABS(n)	Returns the ABSolute value of n
ATN(n)	Returns the ArcTaNgent of n
COS(n)	Returns the COSine of n
EXP(n)	Returns e raised to the power of n
INT(n)	Returns n truncated to an INTeger
LN(n)	Returns natural (base e) logarithm of n
MOD(n1,n2)	Returns the remainder of the division n1/n2
PI	Returns the value of PI
RAND(n)	Sets the seed for a random number
RND(n)	Returns a RaNDom number
SGN(n)	Returns 0 if n is zero 1 if n is positive -1 if n is negative
SIN(n)	Returns SINe of n
SQR(n)	Returns the square root of n
TAN(n)	Returns the TANgent of n

All these functions are fully explained in the keyword section (chapter 6), so we won't devote a great deal of space to them in this chapter. However, it's important to be clear about the circumstances under which these commands can be employed, so let's go back to school and have a brief maths lesson.

In the final analysis it's highly unlikely that you're going to use your MTX's numerical functions unless you've a fairly sound grasp of what COS, SIN and all the rest of them are about. This said, it could be that your memory simply needs jogging, so prepare yourself for a swift numeric jolt.

Let's start with the simplest of the number functions. As the following example demonstrates, it is sometimes necessary to force the MTX to produce a positive value, even if there's a chance of a calculation returning a negative value. Under these circumstances, the ABS function can be applied, since it returns its argument's ABSolute value. In other words ABS returns the number disregarding the + or - sign.

```
10 REM ******************
13 REM ***     ABS     ***
15 REM ******************
20 REM * 500BC - 100AD *
25 REM ******************
30 FOR Y=-500 TO 1000 STEP 150
40 PRINT ABS(Y);
50 IF Y<0 THEN  PRINT "BC" ELSE  PRINT "AD"
60 NEXT Y
```

Another straightforward numeric function, which we've already encountered in circle program is INT. Sometimes we need to ensure that a particular value is always a whole number (or INTeger). The INT function rounds towards zero. Thus:

PRINT INT(7.89) **will display 7**
PRINT INT(-9.57) **will display - 9**

Note that the next lowest INTeger to -9.57 is, of course, -9 and not -10! On the subject of signs, the MTX's SGN function returns a result which indicates the status of the value to which it is applied. For instance, in the statement SGN(x), if x is positive the function will return 1, if x is zero the statement will return 0 and if x is a negative value SGN will produce -1.

We have already encountered the ^ symbol and the way it enables us to raise one value to the power of another. Well, the EXP function raises its argument to the value of e (which is approximately 2.71828183). This is extremely useful if you happen to be performing calculations with the MTX's natural log function (LN), since EXP(LN(n)) returns the antilog of its argument. The LN function itself can be useful when working with very large numbers that might potentially throw up an overflow error. Under such circumstances you can ensure that:

V<7^N

does not throw up an overflow error by making sure that:

LN(V)<N*LN(7)

Although it is possible that V ^ 7 could cause an overflow, the LN formulation performs the test without risking an error report. The Memotech's RaNDom number function is quite complex and explained at length in the keyword section. For now we'll simply outline the MTX's RaNDom number facilities. When a RND statement is executed the computer does not actually return a true RaNDom number but selects a number (known as a seed value), upon which it performs calculations to produce a numerical sequence. Thus, although the numbers generated appear to be random they are in fact 'pseudo-random' numbers. By using the RAND command it is possible to select your own seed number. This enables us to determine a predictable 'random' sequence by using positive RAND values, or unpredictable 'pseudo' random numbers by using negative values. For example, if you RUN the following routine:

```
10 REM ******************
13 REM ***  RAND/RND ***
15 REM ******************
20 RAND (1)
30 FOR A=1 TO 5
40 PRINT RND
50 NEXT A
```

The Memotech will always PRINT out the same RaNDom sequence. However, if you change line 20 to:

```
20 RAND (-1)
```

the sequence of values generated will always be different.

So, if you want to create a random number in the range 0 (which it can equal) to 1 (which it will never quite reach),the format is:

r=RND or **r=(RND*1)**

where r is the random value returned. On the basis of the second format it should be easy to see that you can create random numbers between 0 to 10 by using the following format:

r=(RND* 10)

If we want to set both the upper and lower limits of the random number we must use the following format:

r=(RND*x)+y

where x is the difference between the lower and upper limits plus one, and y is the lower limit. Thus:

INT(RND*15)+5

will return an INTeger number between 5 and 14. The upper limit of this statement is 14 because the INT function truncates the number to produce the next lowest whole number so, the 'random' number will never quite reach 15.

Before we move on to discuss your computer's trigonometric functions, two remaining numeric functions deserve a mention. The first of these is SQR. As you probably know, a square root is the value which, when multiplied by itself, produces the number of which it is the square root. What!! In other words, 4 is the square root of 16, since 4*4=16. The Memotech's SQR function returns the square root of its argument. Thus:

PRINT SQR(16)

will return 4, and:

PRINT SQR(4)

will return 2.

The MTX's final numeric function is MOD, which returns the remainder of the division of its argument. Thus:

PRINT MOD(5,2)

will return 1, since 5/2=2 with a remainder of 1. The following example uses MOD to get around the fact that INT rounds towards zero.

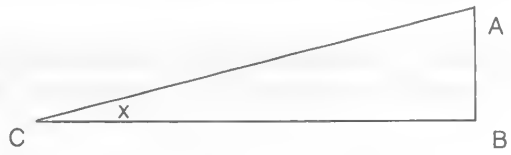
```
10 REM ****************
13 REM ***    MOD     ***
15 REM ****************
20 PRINT "ENTER TWO VALUES (10-100) FOR THE"
30 CSR 10,5: PRINT "EXPRESSION X/Y"
50 CSR 6,10: PRINT "X MUST BE GREATER THAN Y"
60 INPUT X$: LET X=VAL(X$)
70 INPUT Y$: LET Y=VAL(Y$)
80 IF Y>X THEN  GOTO 70
90 LET D=INT(X/Y)
100 IF MOD(X,Y)>Y/2 THEN  LET D=D+1
110 CSR 5,15: PRINT X$;" / ";Y$;" =";D
120 CSR 5,17: PRINT "ROUNDED TO THE NEAREST INTEGE
R"
```

BASIC TRIGONOMETRY

Having dealt (albeit briefly) with your micro's straightforward number functions, we'll now attempt to jog your memory and run through the rudiments of trigonometry in the hope of placing the next sequence of BASIC functions in context.

Your micro has four major trigonometric functions - PI, SIN, COS and TAN. As we have already seen PI in action and it has an entry in the keyword section, we will merely outline its function in this section. It's important to note that PI does not have an argument (see the above list of numeric functions), as it will always return the value of PI (3.14159265...) rounded to eight decimal places.

In order to clarify the values returned by SIN, COS and TAN, we must resort to a painless diagram of a right-angled triangle.



Each of the three functions represents a specific ratio of the sides of a right angled triangle. Taking x to be the angle at point C of the right angle triangle ABC, the various ratios can be expressed as follows:

$$\text{COS(X)} = \frac{\text{BC}}{\text{AC}} \qquad \frac{\text{ADJACENT}}{\text{HYPOTONUSE}}$$

$$\text{SIN(X)} = \frac{\text{AB}}{\text{AC}} \qquad \frac{\text{OPPOSITE}}{\text{HYPOTONUSE}}$$

$$\text{TAN(X)} = \frac{\text{AB}}{\text{BC}} \qquad \frac{\text{OPPOSITE}}{\text{ADJACENT}}$$

The value of understanding this set of relationships is that, given partial information about such a triangle, we can use these formulae to fill in the gaps. For example, suppose we know the angle x and the length of BC. With this information we can work out the lengths of the other two sides by calculating:

```
AC=TAN(X)*BC
AB=AC/COS(x)
```

Whilst the theory might be tedious, would-be graphics programmers will immediately see that these functions are invaluable for future projects. The format for each of these functions is:

```
TAN(x)
COS(x)
SIN(x)
```

When you use any trigonometric function you must remember that, in common with most micros, the MTX does not measure angles in degrees, but in radians. However, since:

360 degrees = 2*PI radians

and the computer returns an acceptable approximation of PI, converting degrees to radians and vice versa is no problem.

conversion to degrees	degrees = radians*2*PI/360
conversion to radians	radians = degrees*360/2/PI

Now suppose we know the lengths of the sides of our triangle, how can we calculate its angles? Well, the MTX's ATN function simplifies such a calculation. ATN(z) returns the ArcTaNgent of its argument (z). The simplest way to consider this function is to regard it as the inverse of TANgent. To make things a little clearer, let's go back to our trusty diagram. Assuming we know the lengths of AB and BC, by using ATN we can now calculate the angle x using the formula:

```
x=ATN(AB/BC)
```

One of the shortcomings of most BASIC dialects is that they do not include a full set of inverse trigonometric functions. However, using the functions we do have at our disposal it is possible to create our own inverse functions. The following formulae simulate arc sine and arc cosine functions:

```
ASN=ATN(BC/SQR(AC-BC*BC))
ACS=PI/2-ATN(AB/SQR(AC-AB*AB))
```

If the prospect of typing in expressions like this every time that they are required in a program fills you with dread, fear not! You can always assign this type of formulae to a variable. Thus:

```
LET ASN=ATN(BC/SQR(AC-BC*BC))
LET ACS=PI/2-ATN(AB/SQR(AC-AB*AB))
```

and providing that all of the variables contained within the formula have been assigned a value there will be no problem.

CHAPTER 4 : LOOPS AND DECISIONS

LOOPS

One of the main strengths of microcomputers lies in their ability to mindlessly repeat a process or a sequence of processes without complaint.

Up until now, we've only used GOTO and GOSUB to create processing 'loops' which allow us to execute a set of instructions more than once. We're now going to introduce an equally important BASIC structure which provides BASIC programmers with a flexible and efficient means of repeatedly executing a sequence of commands. This structure also enables a program to specify how many times the code must be repeated. The structure in question is known as the FOR...NEXT loop.

The importance of the FOR...NEXT loop structure was hammered home to us when we were planning the early chapters of this book. We realised how difficult it was to create any form of meaningful example programs without recourse to this versatile facility. In the following pages we will attempt to illustrate the power of this construction.

Let's start by considering a very simple example. Suppose we want to print out the numbers 1 to 10. Depending on the type of display required, we are faced with a variety of fairly tedious options unless we use a loop structure. If we wanted to print out the numbers across the screen we would either have to use:

```
1 REM ********************************
3 REM *** PRINTING ACROSS SCREEN ***
5 REM ********************************
10 PRINT 1;2;3;4;5;6;7;8;9;10
```

or

```
1 REM ********************************
3 REM *** PRINTING ACROSS SCREEN ***
5 REM ********************************
10 PRINT "1 2 3 4 5 6 7 8 9 10"
```

Or, if we wanted each number on a different line we would have to create an equally clumsy program:

```
1 REM ********************************
3 REM *** PRINTING  DOWN  SCREEN ***
5 REM ********************************
10 PRINT 1
20 PRINT 2
30 PRINT 3
40 PRINT 4
50 PRINT 5
60 PRINT 6
```

```
70 PRINT 7
80 PRINT 8
90 PRINT 9
100 PRINT 10
```

Apart from the fact that this sort of program devours totally unacceptable quantities of memory, it is also hopelessly inefficient and manifestly impractical if you need to generate a display of numbers from 1-1000!

However, a problem such as this is easily resolved by employing the FOR...NEXT structure. Enter and RUN the following example program and then we'll take a look at how it works.

```
1 REM ********************************
5 REM ***   FOR...NEXT LOOP 1     ***
10 REM ********************************
20 FOR A=1 TO 10
30 PRINT A
40 NEXT A
```

On RUNning our example the numbers 1 to 10 will be PRINTed down the screen. The loop statements are contained in lines 20 and 40 and the code to be processed by the loop is the PRINT statement in line 30. As there is only one PRINT statement and the program PRINTs ten numbers to the screen, it should be quite clear that the loop has been executed ten times. How?

Well, when the computer first encounters the statement in line 20 it assigns a value of 1 to the variable A, which is then PRINTed in line 30. The computer continues to process the statements following a FOR... statement until it encounters a NEXT statement. At this point the program loops back to line 20 and repeats the entire process. However, with each pass of the loop the variable A will be incremented by 1. This process continues until A reaches its end value (which in this case is 10).

So, line 20 instructs the Memotech to create a variable called A which is assigned a sequence of values from 1 TO 10. The first value assigned to the loop counter A is determined by the statement's first parameter (in this case, 1). Since there have been no instructions to the contrary, the value of A will be incremented in steps of 1. However, before the value of A can be altered, the computer must execute the statements in the body of the loop (line 30). When it reaches the NEXT statement, the micro checks to see if A has reached its end value (10). If the loop counter does not equal its end value processing recommences from the first statement of the loop where the value of the variable is redefined. When the loop counter (A) reaches its end value (10) processing moves on to the line following the NEXT statement (or in the case of our example program, ends).

The start and end parameters of a FOR...NEXT loop can be integers, floating point values, variables or calculated expressions. It is even possible to assign negative parameters. When the FOR statement adopts the format used in the example above, the value of the loop counter will be incremented in steps of 1 on each pass of the loop. However, it's possible to specify the increment (or decrement) value of the loop counter by adding the keyword STEP to the statement.

```
1 REM  ********************************
5 REM  ***   FOR...NEXT...STEP     ***
10 REM ********************************
20 FOR A=0 TO 100 STEP 10
30 PRINT A
40 NEXT A
```

This program executes line 30 ten times, thus PRINTing 0,10,20,30...100. The loop is only processed ten times (not 100 times) because line 20 tells the Memotech to increment the value of A (0-100) in STEPs of 10 with each pass of the loop.

As with the majority of popular micros, the MTX allows us to omit the loop counter in the NEXT statement. Thus line 40 could read:

```
40 NEXT
```

and the computer would still realise that you actually meant NEXT A. This said, it's often advisable to include the variable name since it make's a program considerably easier to follow. This is particularly true when there is a lot of code between the loop's opening (FOR) and closing (NEXT) statements.

The absence of loop variable names is also confusing when a loop is placed within another loop (or series of loops). When the loops are used in this manner they are referred to as 'nested' loops. Our next example demonstrates the action of such a structure. Note that the inner loop will always be completely processed with each pass of the outer loop.

Since our example is intended to clarify this construction, it's very easy to follow. However, when loops have been nested three of four loops deep and the code within the loop is lengthy and/or complex, listings can become extremely difficult to follow. Therefore it is advisable to use the complete form of the NEXT statement and clearly REM each module.

```
1 REM  ********************************
5 REM  ***       NESTED  LOOPS       ***
10 REM ********************************
15 REM ***     OUTER LOOP START     ***
20 REM ********************************
40 FOR A=20 TO 5 STEP -5
50 REM ********************************
55 REM ***      INNER LOOP START    ***
60 REM ********************************
70 FOR B=5.25 TO 10.25 STEP .25
80 LET X=B*4.3/A
90 PRINT X
100 NEXT B
110 REM *******************************
115 REM ***   END OF INNER LOOP    ***
120 REM *******************************
130 NEXT A
140 REM *******************************
145 REM ***     END OF OUTER LOOP   ***
150 REM *******************************
160 STOP
```

You'll notice that the NEXT statement of the inner loop is coded before the NEXT A. When using nested loops ensure that the NEXT statements of inner loops precede those of the outer loops. If they are coded the wrong way round a 'No FOR' error message will be returned.

You'll come across examples of FOR...NEXT loops throughout the manual, and additional details about these statements can be found in chapter 6.

Having introduced the final loop control structure available to MTX programmers we can now go on to discuss the final method of introducing data into BASIC programs. We have left this explanation of DATA statements until now because they are commonly used in conjunction with FOR...NEXT statements.

READING DATA

In the previous chapter we saw how it is possible to feed the computer with information from the outside world using INPUT and INKEY$ commands, and by assigning string and numeric values to individual variables. DATA statements offer an alternative means of storing data within a program, and is normally employed when large amounts of data need to be accessed and assigned to variables in a specific order.

Storing information in DATA statements is both simple to code and easy to access. The data itself is held in a program line (or a series of lines) which can appear anywhere in a program. This said, DATA statements are traditionally placed at the end of a program. Let's take a look at some typical statements:

```
100 DATA MON,TUES,WED
110 DATA 23,56,2.6
120 DATA THURS,12,FRI,34
```

As you can see from these statements, both string and numeric data can be stored in DATA statements and it is also possible to mix both types within the same program line. You should also notice that string data does not require quotes and that each item of data is separated by a comma. (However, commas cannot be used as DATA items since the comma serves as the DATA separator.) Before we move on to examine how this data is accessed a few words remain to be said about the DATA statement itself.

If the DATA statement is entered in its standard format:

```
100 DATA X,Y
```

it will place a leading space in front of the first item of data. Although this will have no effect on numeric data it will cause problems when using string data, particularly when this data is being used to create screen displays. This means that DATA must be entered in its abbreviated format (D.). You should also ensure that there is no space between the DATA statement and the first data item. Similarly, when EDITing a DATA statement you will need to reduce it to the abbreviated format and delete the space(s) preceding the first data item.

So, how are data items accessed? Well, DATA is always used in conjunction with the READ statement, which must be accompanied by the appropriate type of variable. For example:

```
5 REM  **************
10 REM ***  DATA  ***
15 REM  **************
20 FOR C=1 TO 6
```

```
30 READ A
40 PRINT A
50 READ A$
60 PRINT A$
70 NEXT C
80 DATA 12,RED,14,BLUE,16,CAT
90 DATA 45,MAN,34,NEXT,2,DOG
100 STOP
```

When the MTX encounters a READ for the first time it searches for the first data item (in the first DATA statement) against which it places a data pointer. So, in our example, line 30 READs the DATA statement in line 80 and assigns the value of the first data item (12) to the numeric variable A, which is PRINTed in the following line. The next READ statement (line 50), is accompanied by a string variable, which is just as well since the data pointer has moved on and is now placed against a string (RED). This is then PRINTed out and the program loops back and repeats the process until all twelve items of DATA have been READ and PRINTed.

You must make sure that for every execution of a READ statement there is a corresponding data item, otherwise the program will be halted by an"No data" error report. You can test this by changing line 20 to:

```
20 FOR C=0 TO 6
```

The program will also stop with a 'Mismatch' error if a READ statement with a numeric variable is used to access string DATA. The items held in a DATA statement can only be READ once in the course of a program unless a RESTORE statement is used. Change line 20 back to its original form change line 100 to:

```
100 GOTO 20
```

If you RUN the program it will return a"No data" error report when the FOR...NEXT loop is executed for the second time. Now add:

```
95 RESTORE 80
```

and each time the loop is completed the RESTORE statement sets the DATA pointer back to the first DATA item in line 80 and allows the program to RUN indefinitely. Most micros allow DATA to be RESTOREd, but the MTX's capacity to RESTORE to a specified line number is a valuable enhancement of the statement. Intelligent coding of DATA statements enables programmers in MTX BASIC to access specific segments of DATA statements at appropriate points in a program, which can be controlled by RESTORE ln.

It is possible to include more than one variable in each READ statement. Our example program above could have been written thus:

```
5 REM  ***************
10 REM ***  DATA  ***
15 REM  ***************
20 FOR C=1 TO 6
30 READ A,A$
```

```
40 PRINT A: PRINT A$
50 NEXT C
60 DATA 12,RED,14,BLUE,16,CAT
70 DATA 45,MAN,34,NEXT,2,DOG
80 STOP
```

Needless to say, when using a READ statement with multiple variables it is important to ensure that they are applied to the appropriate type of DATA item.

Used in this way, READ and DATA statements will only store an item in the READ variable, until the next READ statement is encountered. In the example above we are redefining the values of A and A$ with each pass of the loop. So, how can we access all the items of data at any point in the program without creating a different variable for every READ statement? Well, there obviously has to be an answer, otherwise DATA statements might just as well be replaced by line after line of LET statements. The solution lies in another DIMension!

DIMENSIONING AND ARRAYS

You'll be relieved to learn that we're about to outline the final MTX data facility. Assume we want to PRINT out a calendar. Now obviously our display is going to require the days of the week and the months of the year, which will have to be stored as data within the program. We could obviously construct a different PRINT statement for each day and each month, or else introduce a little flexibility by assigning data to variables.

While the latter approach is certainly the best of the two options, it requires the creation of nineteen different variable names (twelve for the months in the year, and seven for the days of the week). Think how much simpler it would be if we could consider all the days of the week as a variable called DAY$ and all the months in a year as a list we could store in a variable called MONTH$. In the best of all possible worlds we could, for example, PRINT out January by telling the computer to PRINT MONTH$(1) and Monday by PRINT DAY$(1). Well, necessity being the mother of invention, precisely such a facility is available in BASIC, and it is known as an array.

The storage of data in arrays is one of the most important tools available to the BASIC programmer. The creation of arrays is a means of allocating a specified amount of memory space for the storage of a list (of numbers or strings). The list itself is known to the computer by a single variable name (the array variable) and each item in the list is assigned a numerical value (the subscript) by which it can be accessed. For example, in the case of our list of days, we could call the array variable DAY$ and each day of the week would be assigned the numbers 1-7, i.e. Wednesday being the third item on the list would be DAY$(3).

So, we now have a variable list of seven elements called DAY$(x) (where x is the number assigned to the day, i.e. DAY$(1)=Monday). The next dimension we must define is the number of characters in the longest string we want to store (in this case, Wednesday), which is nine characters long. When we DIMension string arrays we have to tell the MTX the maximum number of characters that will be used by the elements of the list. This enables the MTX to allocate sufficient space in its memory for, in this instance, a list of seven elements where each element is a maximum of nine characters long. So our DAY$ array variable will be DAY$(x,9). However, it is only necessary to include this second parameter when using string arrays. You do not need to tell the MTX the maximum number of characters of numeric arrays.

Before we can store any data in our DAY$ or MONTH$ lists we have to tell the computer how much memory each list likely to occupy. In other words, what are the DIMensions of the memory space that are going to be given over to either. We do this by using the DIM statement.

Since storing data in arrays plays such an important role in BASIC programming, and the concept of DIMensioning is often confusing to new programmers, we have included a fairly substantial DIM entry in the keywords section. Before continuing with the rest of this chapter, turn to the DIM entry and make sure that you understand the process of DIMensioning an array.

It is important to remember that the DATA command should be entered in its abbreviated format in the manner outlined earlier in the chapter. Let's suppose you DIMension an array to hold nine elements (items of data) which are a maximum of six characters long and then forget the leading space inserted by the DATA statement in front of the first data item. As soon as the computer encounters the offending data item (which is now seven characters long) and tries to store it into the array variable it will return a 'No space' error because the computer had only assigned enough memory space for a maximum of six characters per item of data.

Having established how data is stored in arrays it should be clear that the ability to access items by their subscript enables complex operations to be performed on large quantities of data, using simple code that uses memory space economically.

LOGICAL CONTROL

By now you have hopefully grasped the essential elements of BASIC programming. We've looked at the ways data can be stored and displayed, and how the Memotech's string and numeric functions can be used to process the data. We've also outlined the methods of re- directing control in a program with the use of loops, subroutines and GOTO statements. Now it's time to look at how your Memotech can be programmed to test data and determine the order and nature of processing.

Most computers-in-the-future horror stories centre around the intelligence of machines that take over the world. From what we have seen so far, your computer does not constitute much of a threat as regards world domination. This said, use of IF...THEN...(ELSE) statements makes it possible to create BASIC programs that simulate intelligence.

In our discussion of FOR...NEXT loops we saw how the computer tested the value of the loop counter to see whether it equalled the loop's end value. It is this ability to logically compare values that provides the basis for the IF...THEN...(ELSE) construction and intelligence simulation.

LOGICAL TESTS

In spite of appearances to the contrary, the world seen through the eyes of the MTX is very simple. Something is either True or False, and it has no capacity to come to any other form of conclusion when accessing a condition. BASIC uses this somewhat uncompromising quality to good effect when testing and comparing values in order to determine a course of action.

Like so many BASIC commands, the IF...THEN...(ELSE) statement has a self explanatory format:

IF <condition> THEN <action> ELSE <action>

(The ELSE is not actually entered in brackets, it is bracketted to indicate that it is an optional part of the statement.) The condition after IF can be one of a variety of alternatives. The table below lists the conditional operators that can be used to determine the relationship between values:

= equal to
< less than
> greater than
<= less than or equal to
>= greater than or equal to
<> not equal to

Thus, the IF...THEN...(ELSE) statement is split into three distinct sections. The first part contains the expression that estalishes a relationship between values, and tests for its Truth or Falsity. IF the condition is False the computer will pass over the statement following THEN, and, assuming the ELSE statement has been included, it will execute the commands that follow it. If the ELSE statement was not included control will pass to the next program line.

IF the expression is True, the instructions that follow the THEN statement will be executed and, if applicable, the ELSE statement ignored. The instructions which follow THEN and ELSE can be any legal BASIC statement. For instance:

```
IF X>10 THEN LET Y=15 ELSE LET Y=5
```

is a legitimate structure, or a program can be terminated by:

```
IF D$=A$ THEN STOP
```

One of the most powerful applications of the IF...THEN...(ELSE) construction is when it is used to direct the control of a program:

```
IF V<>2 THEN GOSUB 500
```

If you wish the consequence of a True expression to be the execution of a GOTO statement, you could use the following statement:

```
IF V<=4 THEN GOTO 50
```

One of the more tedious aspects of creating interactive programs in BASIC is that a large percentage of the code must be devoted to guarding against human stupidity. As a general (and fairly inflexible) rule, INPUT statements should always be followed by some form of INPUT check. For example:

```
10 REM **********************
13 REM ***  INPUT CHECK  ***
15 REM **********************
20 CLS : INPUT "ENTER A NUMBER (1-10) ";N
30 IF N<1 OR N>10 THEN  GOTO 20
40 PRINT "CONDITION IS SATISFIED"
```

Just because a user is asked to enter a number between 1 and 10 does not necessarily mean that a value within this range will actually be keyed in. (Unlike computers, humans excell in the contradiction of instructions!) However, the inclusion of line 30 in the last example gives the user of the program very little choice in the matter. If the number entered (N) is less than (<) one OR greater than (>) ten, the program will simply return to the INPUT statement in line 20.

You'll notice that we've used the logical condition operator OR to combine two expressions in the first part of line 30. MTX BASIC provides us with three such operators (the other two being AND and NOT), all of which are fully described under their respective entries in the keyword chapter. For now, we'll simply look at a few examples which demonstrate how they can be used to combine conditions in IF...THEN statements.

BOOLEAN LOGIC

The importance of logical (or Boolean) operators should be clear from our INPUT check example. If the check is to serve its purpose it is obviously essential that both expressions are True before the program continues. Thus, the statement in line 30 uses OR in exactly the same way as the word is used in English: IF

N is either less than one OR if N is greater than ten return to line 20. But we're jumping the gun a little. Let's look at each of these operators in turn, and explain the consequences of the Truth or Falsity of the expressions they combine. The important point to remember is that although more than one expression can be tested by an IF...THEN...ELSE statement, there can only be a single evaluation of the statement as a whole. The testing of logical conditions reveals another difference between micros and men. For the MTX there is no such thing as a half truth!

AND

Having insisted that the MTX can only evaluate a conditional expression as either True or False, it's time to back pedal a little in the interests of precision. Whilst the outcome of a logical test can only be True or False, the computer does not actually have these words in its vocabulary. When asked to pass judgement on an expression,the computer returns either -1 or 0, which are its representation of True and False respectively. You can test this out be entering a couple of statements as direct commands. For example:

```
LET A=5 : PRINT(A>10)
```

will return a 0, since the expression that A is greater than 10 is False(0). Try:

```
LET A=5 : PRINT(A<>10)
```

Since A is clearly not equal to 10, the expression is True (-1) and thus -1 is returned.

So, the computer is obviously capable of logically assessing the Truth or Falsity of an expression. However, it is often necessary to relate a number of expressions to one another in order to discover the logical status of a combined expression. When AND is used to relate expressions the final result will only be True if each element of the statement is True. For example:

```
PRINT (4=4 AND 3=2)
```

will PRINT 0 (False), because although the first part of the expression (4=4) is True, the second part (3=2) is False and thus the expression as a whole is False. When such an expression is used with IF...THEN the complete statement resembles an English sentence whose meaning is quite clear:

```
10 REM **************
13 REM ***  AND  ***
15 REM **************
20 INPUT "PLEASE ENTER A NUMBER ";N
30 IF N>=5 AND N<=10 THEN  PRINT "TRUE" ELSE  PRIN
T "FALSE"
```

Obviously the computer will only get the chance to PRINT out True if the number falls between 5 and 10, ELSE it will PRINT out False. We can also use this sort of construction to relate string expressions:

```
100 IF S$="F" AND M$="Y" THEN  LET A$="MRS"
```

The traditional (and indeed clearest) method of describing the effects of combining expressions is by the means of a Truth table. What follows is a Truth table for AND, in which the result listed under the headings Condition 1 and Condition 2 define the logical status of the component expressions relate by AND:

Condition 1		Condition 2		Result
TRUE	AND	TRUE	=	TRUE
TRUE	AND	FALSE	=	FALSE
FALSE	AND	TRUE	=	FALSE
FALSE	AND	FALSE	=	FALSE

As the table graphically demonstrates, every element of a multiply conditional expression whose components are related by AND must be True before the expression as a whole can be deemed True (-1).

OR

OR is used as a relational operator when a conditional statement must return True if any one OR all of its component expressions are True. Thus:

```
100 IF A=10 OR B=-1 THEN  GOSUB 450
```

will pass control to line 450 if either A=10 or B=-1 OR if both conditions are True. As with AND, OR can also be used to test conditions relating expressions using strings:

```
200 IF A$="Y" OR B$="N" THEN  GOTO 10
```

So, the Truth table for the OR operator is:

Condition 1		Condition 2		Result
TRUE	OR	TRUE	=	TRUE
TRUE	OR	FALSE	=	TRUE
FALSE	OR	TRUE	=	TRUE
FALSE	OR	FALSE	=	FALSE

Thus OR ensures that a conditional statement in which it appears is only False if all its component expressions are False.

NOT

This is neither the time nor the place to embark upon the perennial debate on whether NOT should be used at all as a relational operator. Suffice it to say that NOT should always be used with care since its use does not always produce such immediately predictable results as its logical colleagues AND and OR. Essentially, NOT inverses the logical status of the expression it precedes. However, whilst NOT (False) will always return True, NOT (True) will not always return False. Let's take a look at NOT's disconcertingly simple Truth table:

NOT (expression TRUE) = FALSE
NOT (expression FALSE) = TRUE

So when used with conditional BASIC statements the effects of the three relational operators is quite straightforward. Indeed IF...THEN statements combining expressions with the operators are close enough to their counterparts in English to be virtually self explanatory.

Take some time out to construct some demonically complex expressions using one or more of the operators. Use the Truth tables if you are at all surprised by the results the computer returns.

CHAPTER 5 : SAVED FOR POSTERITY

OFFLINE STORAGE

The MTX will only store a program for as long as the computer is powered up. In other words, whenever you turn the machine off you'll lose the contents of it's memory. It's obviously impractical re- entering a program each time you want to use it, which is why we need an external or 'off-line' method of storage.

Programs can be stored on either cassettes or disc and in this chapter we will be looking at cassettes, the most popular of the two methods. It should be stressed that cassettes are not the most efficient or reliable method of storing data. Discs, on the other hand, although fast and easy to use are very much more expensive.

Programs are stored on cassette in much the same way as anything else is recorded on tape, except that we're interested in storing data rather than music or speech. When you initiate the recording process a data signal is transmitted from computer to cassette player, and stored on the magnetic tape of an ordinary cassette. When you want to re-load the program into the computer's memory you simply inform the MTX of your intentions, replay the cassette and the program will be reconstructed in much the same way as a song is recreated on your hi-fi.

Before we look at the keywords and connection which access the MTX's cassette handling facilities, let's take a quick look at the best type of cassette player to use with your computer. The cassette lead that comes with the MTX has two plugs at each end. So the first thing to bear in mind if you intend buying a dedicated cassette player is that it should be one with 'EAR' and 'MIC' sockets (not all cassette recorders have this facility). However, if you already own a cassette recorder and it doesn't have these ports, don't fret, you won't have to rush out and buy a new one. Most hi-fi and computer stores stock a wide range of leads and you should be able to find the right one without too much difficulty. Even if they don't have the lead you're looking for they can probably be persuaded to make one up for you.

Fortunately, the best sort of recorder for the job doesn't require an enormous financial outlay. Surprisingly enough, computers are not over-fond of high-quality stereo recorders, since such equipment tends to pick up a variety of extraneous noises which confuse the micro and lead to rejected recordings. There's no harm trying it out whatever system you own, but if you use a stereo you must switch it to mono before you SAVE or LOAD a program.

When buying a cassette recorder there are a number of essential points to watch out for. Firstly, it's useful to find one with a tape counter so that you can make a note of where one program ends and another begins. This will enable you to mark your cassettes clearly and trace any programs without ploughing through miles of tape. It's worth searching out a recorder with both volume and tone controls. Computers are fussy about the type of signal they receive, and it helps to be able to regulate this by adjusting volume and tone levels.

You should always try to stick to the same cassette recorder when SAVEing and LOADing programs, since the alignment of record and playback heads varies from one machine to another. These days, most of the big chain stores sell micro-dedicated cassette recorders (known as data recorders) for around 30 pounds. As they have been specially designed to deal with computer data they are normally a worthwhile investment. As we mentioned earlier, it's very important to clean the heads on your recorder on a regular basis. By neglecting to take this simple precaution and allowing the heads to get dirty you'll find it increasingly difficult to LOAD or SAVE anything.

Before we move on to look at how this method of storage is accessed let's take a quick look at the cassettes themselves. Never try to save money when buying data cassettes as it will inevitably prove to be a false economy.

It is always advisable to buy short tapes such as C12's, C15's or C10's. If you use anything longer, you'll suffer the obvious irritation of having to wade through twenty programs to find the one you want, and risk the added danger of losing a large number of programs should the tape become damaged. Apart from these two very sound reasons for not using this type of cassette, you'll also find that these longer cassettes have a tendency to stretch, thus corrupting your programs.

CONNECTING THE SYSTEM

Having established the dos and don'ts of buying a cassette player, let's take a look at how to use it. The first thing we need to do is to connect the cassette player up to the MTX. Take a look at the cassette lead. There are two pins at each end, one is black and the other grey. Plug one into the socket marked 'MIC' on the back of the MTX and insert the same coloured pin at the other end of the lead into the socket market MIC on the recorder. Plug the other pins into the sockets marked 'EAR'. Note that the same coloured pins must be plugged into corresponding sockets.

Your computer uses three commands to deal with cassette handling: LOAD, SAVE and VERIFY. We'll examine each of these in turn, starting with LOAD. Predictably enough, this is the command which LOADs programs into the computer. It is normally entered as a direct command and can use one of the two formats:

LOAD ""

or

LOAD "filename"

The first of these formats is used when you don't know the name of the program that you are trying to LOAD, or when you want to LOAD the first program on a tape. If you try to LOAD a program and use an inaccurate filename the MTX will not recognise it and consequently won't LOAD the program, so this is the syntax to use when your unsure about a filename.

The second format can be used when you're sure about the name of the program, It's important to note that you must use the same name by which the program was SAVEd, so make sure that your cassettes are accurately marked. The computer distinguishes between upper and lower case filenames, so if you SAVE a file called E.G. and then ask it to LOAD "e.g." the MTX will be unable to locate the file you're looking for.

Insert one of your free games programs into the cassette player and type in LOAD "", press <RET> and then press the PLAY button on the cassette player. The MTX will now LOAD the first program that it encounters on the tape. When it finds a program it will display the message:

FOUND: filename
LOADING

and once the program has been LOADed the 'Ready' prompt will be displayed on the screen. If you have any problems LOADing a program (either nothing will appear to be happening or a BK error message returned) try altering the volume and tone controls on the cassette player and then attempt to LOAD the program again. If you still have problems, it's quite possible that the tape has been 'corrupted'. This means that your recording of the program has , for one reason or another, been damaged and the MTX will not accept it as valid input.

SAVEING

As cassettes are not a particularly reliable method of storage it is always advisable to make at least two copies of each program, preferably on separate cassettes. SAVEing a program involves transferring data from the MTX to a cassette. Try typing in one of the example programs in this book and then SAVE it using the format below. (Make sure that your cassette lead is still plugged into the appropriate sockets.)

SAVE "EXAMPLE"

Before pressing <RET>, place the cassette in the recorder, making sure that the the non-magnetic 'header' of the tape is past the heads, since you cannot store any information on this part of the tape. Now, press the RECORD and PLAY buttons on your recorder and press <RET> on the MTX. When you SAVE a program always make sure that you give it a sensible filename (one you'll remember) and then clearly label the cassette.

VERIFYING

Once you've SAVEd a program it is a good idea to check that the data transfer has been successful by VERIFYing a program. Rewind the tape, and type in:

```
VERIFY "EXAMPLE"
```

or

```
VERIFY ""
```

press <RET> and the press the PLAY button on the cassette recorder. The MTX will then compare the program on tape with the program in memory and, if the two match up it will display the message:

```
FOUND: filename
VERIFYING
```

followed by the 'Ready' prompt. However, if the two programs don't match you'll be presented with a 'Mismatch' error message. Should this happen you will have to SAVE and VERIFY your program again, repeating the process until the MTX is convinced that the two versions are identical.

One final point that should be noted when VERIFYing a program is that it is not possible to BReaK out of the process unless the tape is actually running.

FLEXIBLE SLICES

Having gone to some lengths to introduce our readers to the wonders of BASIC string manipulation, this is as good a time as any to reveal that Memotech BASIC offers string-slicing facilities which to all intents and purposes make the commands we have just discussed redundant! This said, although your MTX has string-handling capabilities considerably in advance of those offered by other home computers, you should familiarise yourself with the action of MID$, LEFT$ and RIGHT$ simply because they provide the basis for string manipulation in most dialects of BASIC.

So what's so special about the way the Memotech handles strings? Well once a string variable has been defined, any substring can be directly accessed via the variable itself. In other words, we can create substrings without having to utilise any of the traditional string-handling statements.

Let's take a look at some simple examples that will hopefully add substance to these apparently wild claims. We'll kick-off by assigning a string to a string variable in the usual manner:

```
LET A$="MEMOTECH"
```

Now let's assume that we want to extract the substrings "MEMO" and "TECH". On most micros such operations would require the use of LEFT$ and RIGHT$ statements:

```
PRINT LEFT$ (A$,4) will return "MEMO"
PRINT RIGHT$ (A$4) will return "TECH"
```

However, Memotech BASIC allows direct access to substrings:

```
PRINT A$ (1,4) will return "MEMO"
PRINT A$ (5,4) will return "TECH"
```

This method of string-slicing can also be used to generate single character substrings. For example:

```
PRINT A$(4) will return "O"
```

We can also simulate MIDS. The following routine:

```
10 LET A$="MEMOTECH"
20 FOR A=1 TO 8
30 PRINT MID$ (A$,A,1);"";
40 NEXT A
```

Could be coded with the following modification:

```
30 PRINT A$ (A,1);"";
```

Simplicity itself! The only point to be borne in mind when using such constructions is that they can only be applied to standard string variables. If you want to use array variables in this manner the individual elements of the array in question will have to be assigned to a transient variable.

CHAPTER SIX: THE KEY TO BASIC

Now that we have completed our schematic introduction to the logic and constructions available in MTX BASIC, we shall endeavour to systematically detail each of the language keywords (or "reserved" words).

The keywords are presented alphabetically in a standard format. Each entry comprises a format example using the following standard symbols:

V=	numeric variable name
A$	= string variable or expression
i	= integers or whole numbers
n	= floating point or decimal value
c	= conditional logical expression
ln	= program line number
addr	= a memory location address
x,y	= represent screen coordinates

Any special symbols required to clarify particular keywords will be defined within the entry in question. When appropriate, we have included a short example program designed to show the keyword in action and a list of related keywords which should be referred to as a means of situating commands in their group context.

ABS
SYNTAX : v=ABS(n)

ABBREVIATION : AB.

Numeric function which is used to ensure a positive value result from a numeric variable or expression.

There are occasions when it is essential to ensure a positive result from a calculation, or else non-negative data from user input. In short, there are programming situations in which we must avoid a negative value, and require what is known as an ABSolute value. For obvious reasons, there are certain circumstances in which a negative value will cause a program to crash, and ABS is the function that guards against such an event.

For example, suppose we wanted to establish CSR (cursor) parameters for the display of user generated input. Obviously the appearance of negative data in such input would require an additional character ("-") whose inclusion could disrupt an otherwise tidy display. By using ABS we can get around this problem as our example program demonstrates.

The simplest way of understanding the value of this function is by considering the following: If we ask the computer to consider the expression x-y, the result will obviously only be positive as long as x is greater than y. However, ABS(x-y) will always return a positive result, regardless of the values of the two variables. Thus:

```
50 PRINT ABS(X-Y)
```

when x=100 and y=150 will return a positive 50, although the result of the actual calculation will, of course, be -50.

The following example uses ABS to ensure a tidy tabular display, regardless of whether the values entered are negative or positive.

```
10 REM ************
20 REM ***  ABS2 ***
30 REM ************
40 CSR 10,3: PRINT "INPUT FIVE NUMBERS"
50 CSR 8,4: PRINT "(POSITIVE & NEGATIIVE)"
60 FOR A=1 TO 5
70 CSR 10,7+A: PRINT "NUMBER ";A;" IS"
80 GOSUB 200
90 IF N<0 THEN  CSR 24,7+A: PRINT "-"
100 NEXT A
110 STOP
200 CSR 2,20: INPUT N
210 CSR 2,20: PRINT CHR$(5)
220 CSR 24,7+A: PRINT ABS(N)
230 RETURN
```

As well as returning positive values, ABS also rounds any values to which it is applied to eight decimal places. Thus:

PRINT ABS(23.123456789) will return: 23.12345679

RELATED KEYWORDS: SGN

ADJSPR
SYNTAX : **ADJSPR p,n,v**

ABBREVIATION : **AD.**

ADJusts a value previously assigned by the SPRITE or MVSPR commands.

The ADJSPR command only alters a single parameter at a time, thus increasing the speed of program execution. The variable n is the sprite number (1-32) and v is the new value to be assigned to the parameter chosen by p.

VALUE OF P	MEANING	RANGE OF V
0	Sprite pattern	0-31 for 16*16 sprites 0-127 for 8*8 sprites
1	sprite colour	0-15
2	position x co-ordinate	0-255
3	position y co-ordinate	0-255
4	speed x direction	0-255 128-255 gives leftward movement
5	speed y direction	0-255 128-255 gives downward movement

The example below demonstrates how you can move sprite 1 across the screen from left to right.

```
30 REM ** PUT SPRITE ON SCREEN **
50 VS 4: CLS
60 CTLSPR 2,1
80 GENPAT 3,0,60,126,219,255,231,126,36,60
90 SPRITE 1,0,127,96,0,0,1
110 REM ** MOVE SPRITE ACROSS SCREEN **
130 FOR T=0 TO 255
140 ADJSPR 2,1,T
150 NEXT T
160 GOTO 50
```

RELATED KEYWORDS: CTLSPR, GENPAT, SPRITE, MVSPR, VIEW

AND
SYNTAX : c AND c
: e AND e
ABBREVIATION : none

A logical operator that performs logical AND operations on two expressions.

AND is one of the MTX's Boolean or logical operators. It is used to determine whether a condition is True or False. AND works in much the same way as it's used in the English language, acting as a conjunction to combine logical expressions. To produce a True result, both of its conditions must be met. If only one or neither of them are met the condition will be False.

Condition1	Conditional operator	Condition2		C1 AND C2
TRUE	AND	TRUE	=	TRUE
FALSE	AND	TRUE	=	FALSE
TRUE	AND	FALSE	=	FALSE
FALSE	AND	FALSE	=	FALSE

When AND is used in an IF...THEN statement it will only activate the THEN instruction if the results of both conditional expression(s) are True, otherwise processing will either move on to the following line or the instructions following the optional ELSE statement.

```
10 REM *************
13 REM ***  AND  ***
15 REM *************
20 INPUT "WHAT IS YOUR NAME? ";N$
30 INPUT "ARE YOU MALE OR FEMALE? (M/F) ";SEX$
40 INPUT "ARE YOU MARRIED OR SINGLE? (M/S) ";B$
50 IF SEX$="F" AND B$="S" THEN  LET NAME$="MISS "
ELSE  LET NAME$="MR. "
60 IF SEX$="F" AND B$="M" THEN  LET NAME$="MRS. "
70 PRINT "HELLO ";NAME$;N$
```

The above program will only PRINT MISS if both the conditions in line 50 are met (i.e. if you indicate that you are a female (F) and that you are single (S)). Similarly, it will only PRINT MRS if you both the conditions in line 60 are met (i.e. that you are both female (F) and married (M)), otherwise it will PRINT out MR.

RELATED KEYWORDS: OR, NOT

ANGLE
SYNTAX : **ANGLE x**
ABBREVIATION : **ANG.**

Sets the initial drawing direction when using the MTX's 'turtle graphics' type commands.

The value of x is measured in radians and specifies the initial drawing direction. It starts at x=0 rad. which is pointing horizontally to the right and moves anti-clockwise as the value of x increases up to PI*2 (also facing to the right). If x is larger that PI*2 it is treated as x-PI*2. Even though it is permissable to use decimal numbers and/or variables (eg LET Z = 1.345: ANGLE Z) in the statement's argument, it is generally easier to use fractions or multiples of PI. (Refer to chapter 9 for a radian to degree conversion chart.)

The statement below will set the initial draw direction to face downwards:

ANGLE PI*3/2

For further information on this command refer to chapter 9 where it is discussed at length.

RELATED KEYWORDS: PHI, DRAW, ARC

ARC
SYNTAX : **ARC <length>,<angle>**
ABBREVIATION : **AR.**

Draws an arc of a circle starting at the current plotting position.

The initial drawing direction of the ARC command is determined by the angle value currently stored in the computer. Both the plotting position and direction are updated by this command. The angle parameter determines the curvature of the arc by specifying what angle is subtended - the larger the angle, the tighter the curve.

The example program below draws an ARC of a circle with a length 50 and an angle of 90 degrees (PI/2 rad.).

```
1 REM  ************
3 REM  ***  ARC  ***
5 REM  ************
10 VS 4: CLS
20 PLOT 120,90
30 ARC 50,PI/2
40 GOTO 40
```

For furtner information on this command refer to chapter 9 where it is explained at length.

RELATED KEYWORDS: ANGLE, PHI, DRAW

ASC
SYNTAX : **v=ASC("A")**
: **v=ASC(A$)**
ABBREVIATION : **none**

A function which returns a number between 0 and 255 which corresponds to the ASCII code of the first character of the statement's argument.

The MTX does not store its characters as they are seen on your screen, but holds them as numbers known as ASCII codes. ASCII (pronounced as- key) stands for American Standard Code for Information Interchange, and each character in the MTX's character set is identified by its own unique ASCII number. Hence, the letter R is not stored as the letter itself, but as its ASCII code equivalent which is 82. ASC complements CHR$, which returns the character represented by the code of its argument. If ASC is applied to a null string -1 will be returned.

The following program demonstrates the role of ASC in input checks. Line 40 tests to see whether the code of each character that has been entered falls between the range 48-57, which represent the numbers in the MTX's character set. Unless the input is a number, it s ASCII code will not be printed to the screen.

```
10 REM ************
13 REM ***  ASC  ***
15 REM ************
20 CLS
30 INPUT "ENTER ANY STRING ";A$
35 CLS
40 IF ASC(A$)>=48 AND ASC(A$)<=57 THEN  CSR 10,10:
 PRINT A$
45 PAUSE 2000
50 GOTO 30
```

RELATED KEYWORDS: CHR$, STR$, VAL

ASSEM
SYNTAX : ASSEM In
ABBREVIATION : A.

The command used to enter the Assembler.

To invoke the Assembler you must enter ASSEM followed by a BASIC line number. The statement must be entered as a direct command (it cannot be used within a program). On pressing the <RET> key the word CODE appears at the specified line number. Assemble > appears at the bottom of the screen and by pressing the <RET> key once again you can start to insert your ASSEMBLY code.

At this stage the screen should look something like:

8007 RET (MTX 500) or **4007 RET** (MTX 512)

The numbers above represent the hexadecimal address that the Assembler has furnished for you and the RET is the instruction that currently occupies it. It is now possible to overwrite this with your own instruction, if it is not overwritten it will RETurn you to BASIC.

Each carriage <RET>urn gives a new address, to exit insert mode you can type CLS (followed by the <RET> key) and to exit Assembler altogether and thus return to BASIC you must enter CLS <RET> once again.

For further clarification on this command refer to chapter 12.

RELATED KEYWORDS: CODE

ATN
SYNTAX : **v=ATN(n)**
ABBREVIATION : **none**

The value returned is given in radians, and will fall between -PI/2 and PI/2.

Like all the trigonometric functions on the MTX, ATN is, of course, invaluable when attempting any serious graphical work on the machine. It can be regarded as the inverse of the TAN function, and can be used to simulate inverse functions to complement the MTX's other trigonometric functions (COS and SIN). For a further explanation see chapter 3. Note that the value returned by this function is always measured in radians. To convert radians to degrees simply multiply the value by 180/PI (since 2*PI radians = 360 degrees).

ATN(x) = ARCTAN(x)
ATN(-x) = ARCTAN(-x)

The function **F = ATN(X) is valid for -1<X<Rmax**

where Rmax is the maximum +ve real number available under MTX BASIC.

For - Rmax <X<Rmax one should use the function

F=SGN(X)*ATN(ABS(X))

RELATED KEYWORDS: COS, SIN, TAN

ATTR
SYNTAX : **ATTR p,n**
ABBREVIATION : **AT.**

The ATTR command is used to alter the effect that the other graphics commands have upon the graphics screen. The ATTRibutes are not exclusive, but can be used in any combination. The value of n can be either 1=ON or 0=OFF, turning ON or OFF the specified ATTRibute. The parameter p defines the following:

p	EFFECT
0	Inverse PRINT
1	Over PRINT
2	unPLOT
3	over PLOT

The example PRINTs text on the screen, first in normal type and then inversed:

```
1 REM ***************
3 REM ***  ATTR  ***
5 REM ***************
10 VS 4: CLS
20 CSR 5,5: PRINT "NORMAL PRINTING"
30 ATTR 0,1
40 CSR 5,7: PRINT "REVERSE PRINTING"
50 ATTR 0,0
60 CSR 5,9: PRINT "BACK TO NORMAL AGAIN"
70 GOTO 70
```

For further information on this command refer to chapter 9 where it is discussed at length.

RELATED KEYWORDS: PLOT, LINE, PRINT, CIRCLE

AUTO
SYNTAX : **AUTO ln,i**

ABBREVIATION : **AU.**

Command which activates the computer's automatic line numbering facility.

As our introduction to BASIC established, when developing a program it is always wise to increment the line numbers in steps of ten to facilitate the inclusion of any omissions. Thus a regular, graduated system of line numbering is standard programming practice, and the AUTO line numbering feature simply AUTOmates the process.

So how does AUTO work ? Well, let's take the most common programming situation. Your screen is clear and you're about to create a program. You want your creation to start at line 10 and progress in steps of ten. You must enter the following as a direct command:

AUTO 10,10

Once this statement has been executed, the computer will increment line numbers in steps of ten each time you press the return key. Try it. You don't actually have to enter any code. Enter the statement above and repeatedly <RETURN>. You'll generate a sequence of line numbers starting at line 10 and rising in steps of ten.

AUTO's first parameter establishes the start point for the sequence of line numbers. Many programmers start their program development by coding subroutines. So if your first subroutine starts at line 100, the above statement would have to read:

AUTO 100,10

Thus AUTO's first parameter establishes the point at which specified line increments commence, and the second determines the size of the increment.

You can choose any start point and increment value as AUTO's parameters, although for obvious reasons non-integer values will generate an error message. To return to user controlled line numbering simply enter CLS (bottom right on the numeric key-pad), and press <RETURN>.

RELATED KEYWORDS: none

BAUD
SYNTAX : **BAUD c,r**
: **BAUD r**

ABBREVIATION : **B.**

Input/Output statement which facilitates the selection of the baud rate (r) for the specified RS232 output channel (c). If the statement's first parameter (c) is omitted, the specified baud-rate will be established for both channels.

When transferring data via the computer's RS232 output port, the speed of transfer is set by a BAUD statement. A baud is a term used to measure the transfer of data, calculated in bits-per-second. Thus the higher the baud rate. the faster the data transfer.

The MTX has two RS232 channels, which are identified by the channel codes 0 and 1.

The MTX offers twelve legal baud rates, the slowest being 75 and the fastest 19200. The following table lists the available rates. If you try to use any other values as rates, an "Out of range" error will be generated.

LEGAL BAUD RATES	
75	1200
110	2400
150	4800
300	9600
600	19200

RELATED KEYWORDS

CHR$
SYNTAX : **A$=CHR$(n)**

ABBREVIATION : **CH.**

A string function which converts the ASCII code into the character (or display operation) specified by its argument (n).

CHR$ is the string function which complements ASC. While the latter returns the ASCII codes of a string character, CHR$ performs the reverse operation. For example:

```
10 REM ****************
13 REM ***  CHR$ 1  ***
15 REM ****************
20 LET A=ASC("A")
30 PRINT A
40 LET A$=CHR$(65)
50 PRINT A$
```

Line 20 assigns the code for the character 'A' to the numeric variable A, which is then PRINTed in line 30. Thus 65 is displayed on the screen. Line 40 then uses CHR$ to convert 65 into the character it represents, and thus 'A' is PRINTed in line 50. The following program uses CHR$ to display the computer's upper-case alphabet.

```
10 REM ****************
13 REM ***  CHR$ 2  ***
15 REM ****************
20 FOR A=65 TO 90
30 PRINT CHR$(A);", ";
40 NEXT A
```

As well as providing access to the MTX's standard character set, CHR$ can also be used to control facilities such as cursor movement. This said, the following ASCII codes cannot be PRINTed to the screen as they return the 'SE.B' error message: 1, 2, 6, 14, 15, 16, 17, 18, 19, 27. The following example uses CHR$ to PRINT some familiar lower-case characters to the screen:

```
10 REM ****************
13 REM ***  CHR$ 3  ***
14 REM ****************
20 FOR A=0 TO 7
30 READ V
40 PRINT CHR$(V)
50 NEXT A
100 DATA 109,101,109,111,116,101,99,104
```

Adding the following lines will use CHR$(12) to clear the computer's screen each time the program is repeated:

```
15 PRINT CHR$(12)
70 RESTORE 100
80 GOTO 15
```

There is a full list of ASCII codes in Appendix 4 . A little experimentation with CHR$ will reveal the value of this flexible function.

RELATED KEYWORDS: ASC, STR$, VAL

CIRCLE
SYNTAX : **CIRCLE x,y,r**

ABBREVIATION : **CI.**

Draws a circle on the graphics screen at position x,y with radius r.

The circle must be fully drawn within the confines of the screen otherwise an error report will be generated. As with all graphics commands, CIRCLE cannot be used on the text screen. The example program draws a circle whose centre lies at the middle of the screen, with radius 50:

```
1 REM ****************
3 REM ***  CIRCLE  ***
5 REM ****************
10 VS 4: CLS
20 CIRCLE 127,96,50
30 GOTO 30
```

For further information on this command refer to chapter 9 where it is explained at length.

RELATED KEYWORDS: LINE, PLOT

CLEAR
SYNTAX : **CLEAR**
ABBREVIATION : **CLE.**

Command whose execution voids all variables currently in memory.

CLEAR can be used in a program or entered as a direct command. Once a CLEAR statement has been executed, all variable assignments are wiped from the computer's memory and any attempt to utilise a previously defined variable will result in an "Undefined" error report.

The difference between CLEAR and NEW is that while the latter wipes the sections of memory that store variables and program lines, CLEAR leaves the program intact and only voids the variables.

The example program below demonstrates the action of CLEAR. If you RUN the complete version of the program, processing will halt at line 90 with an 'Undefined' error message. This is because although the arrays V$ and N have been defined by the READ loop (lines 30-60), the CLEAR statement in line 70 wipes all knowledge of this assignment from memory. As far as the computer is concerned, the variables in question simply don't exist! If you RUN the program for a second time, this time deleting line 70, you'll see that the computer will repeat the PRINT sequence without complaint.

```
1 REM ****************
5 REM ***  CLEAR   ***
10 REM****************
20 DIM V$(3,10),N(3)
30 FOR A=1 TO 3
40 READ V$(A),N(A)
50 PRINT N(A);V$(A)
60 NEXT A
70 CLEAR
80 FOR B=1 TO 3
90 PRINT N(B);V$(B)
100 NEXT B
200 DATA MEMOTECH,1,MICRO
210 DATA 2,POWER,3
```

RELATED KEYWORDS: LET, NEW, RUN

CLOCK
SYNTAX : CLOCK A$

ABBREVIATION : CLO.

Assigns start value to the MTX's built-in system clock, which is PRINTed by the TIME$ string function.

TIME$ is the string form of a system variable, and enables us to PRINT out or otherwise utilise a real-time value, whose start point is established by CLOCK.

By using CLOCK in conjunction with TIME$ we can access the computer's hundred hour clock, which operates in much the same way as any other digital clock. TIME$ stores a system variable, which "translates" the computer's internal clock into the hours, minutes and seconds used by humans. If you switch your computer off and then on again and then enter the following as a direct command:

PRINT TIME$

your computer will display the time that has elapsed since power-up. CLOCK enables us to establish a start

point to which the values generated by the computer's system clock are added. Thus the string variable CLOCK allows us to use TIME$ as a means of simulating a real clock. If we want the value of TIME$ to start from scratch each time a program is run we use the statement:

```
10 CLOCK "000000"
```

On the other hand, if we want to set the internal clock to 12:50 we must use:

```
10 CLOCK "125000"
```

In other words, 12 hours, 50 minutes and 00 seconds. The following example program uses CLOCK and TIME$ as a stop watch.

```
10 REM ************
13 REM *** CLOCK ***
15 REM ************
20 CLOCK "000000"
30 LET X=2
40 FOR A=1 TO 35
50 LET SE$=RIGHT$(TIME$,2)
60 LET CO=INT(RND*900)+10
70 CSR 2,2: PRINT "00:";SE$
80 CSR X,12: PRINT "*"
90 PAUSE CO
100 CSR X,12: PRINT " "
110 LET X=X+1
120 NEXT A
130 PRINT "THE * TOOK ";SE$;
140 PRINT " SECS TO CROSS THE SCREEN"
150 PAUSE 3000: CLS : PRINT "AGAIN? (Y/N) "
160 LET D$=INKEY$: IF D$="" THEN  GOTO 160
170 IF D$<>"Y" THEN  STOP
180 CLS : GOTO 10
```

RELATED KEYWORDS : TIME$

CLS
SYNTAX : CLS

ABBREVIATION : C.

The command used to clear the display screen.

When using this command on the text screen or virtual screen then the screen will always clear. When using the graphics screen, the screen will not clear if certain of the ATTRibutes are set (see ATTR). In particular, when using ATTR 3,1 the screen will not clear, but the screen colours can be changed globally by using CLS.

The example program below first prints text onto the screen, and then CLearS the screen again.

```
1 REM **************
3 REM ***  CLS  ***
5 REM **************
10 VS 4: CLS
20 FOR T=1 TO 20
30 CSR T,T: PRINT "HELLO WORLD!"
40 NEXT T
50 PAUSE 1000
60 CLS
70 STOP
```

RELATED KEYWORDS: none

CODE
SYNTAX : **n/a**

ABBREVIATION : **n/a**

The word inserted by the computer into a listing to indicate the start point of Assembler code.

It is important to stress that CODE is not a command and it cannot be typed in directly from the keyboard. It is invoked by the ASSEM command and the computer inserts the word CODE in the line number specified by the aforesaid command.

For example, if ASSEM 30 was entered the word CODE would be inserted at line 30 in the BASIC listing. Its sole function is to indicate that the following lines are written in the Assembler language.

RELATED KEYWORDS: ASSEM

COLOUR
SYNTAX : **COLOUR p,n**

ABBREVIATION : **COL.**

The COLOUR command sets the colours for the graphics screen.
The colour is defined by n, see Appendix 5 for the colour numbers. The value of parameter p selects the type of display to be changed.

p	meaning
0	PRINT PAPER colour
1	PRINT INK colour
2	PLOT PAPER colour
3	PLOT INK colour
4	BORDER colour

The example program PRINTs text in different colours on the graphic screen.

```
1 REM ****************
3 REM ***  COLOUR  ***
5 REM ****************
10 VS 4: CLS
20 FOR T=1 TO 15
30 COLOUR 1,T
40 CSR 5,T: PRINT "THIS IS COLOUR ";T
50 NEXT T
60 PAUSE 5000
70 STOP
```

RELATED KEYWORD: PRINT, ATTR, PLOT, LINE, CIRCLE

CONT
SYNTAX : **CONT**

ABBREVIATION : **CO.**

CONTinues the processing of a program that has been halted by the STOP command or the BReaK key.

When developing a program it is often important to know the value of a particular variable at specific points in its execution. By using the BRK key the program can be halted, the values checked and processing re-started using the CONT command. The MTX will re-commence processing at the precise point at which it was interrupted.

CONT can also be used to re-start a program that has been halted by a STOP statement. Once again, processing will continue from the line following the STOP statement. However, you cannot CONTinue a program if any alterations have been made.

RELATED KEYWORDS: STOP, RUN, GOTO

COS
SYNTAX : **v=COS(n)**

ABBREVIATION : **none**

Returns the cosine of its argument n.

It should be remembered that the computer's trigonometric functions all return values measured in radians and that in order to calculate the angle in degrees, v must be multiplied by 180/PI. (See chapter 9).

RELATED KEYWORDS: ATN, SIN, TAN

CRVS
SYNTAX : **CRVS n,t,x,y,w,h,s**

ABBREVIATION : **CR.**

Defines a virtual screen within the main display screen.

The seven parameters used by the CRVS command are used to determine the following attributes:

n	Virtual screen number (0-7)
t	Type of screen: 0=text, 1=graphic.
x	Position of the top left corner, x co-ordinate.
y	Position of the top left corner, y co-ordinate.
w	Width of the virtual screen in characters.
h	Height of the virtual screen in lines.
s	size of the full screen length : 40 for text screen. : 32 for graphic screen.

The example line creates a virtual text screen of size 10 characters by 6 lines positioned so that the top left corner is at position 10,9. The screen is given the identification number 2 and can then be invoked using VS 2:

```
10 CRVS 2,0,10,9,10,6,40
```

Screen types cannot be mixed. In other words, Virtual text screens cannot be placed upon the main graphic screen and Virtual graphics screens cannot be placed on the text screen.

For further information on this command refer to chapter 9 where it has been explained at length.

RELATED KEYWORDS: VS

CSR
SYNTAX : CSR x,y

ABBREVIATION : CS.

Places a character CurSoR at the specified position.

The argument x describes the horizontal character position, whilst y describes the vertical line position. Any subsequent PRINTing or INPUTting will take place at the new cursor position.

The example below places the CurSoR in the middle of the screen prior to PRINTing the text:

```
1 REM ************
3 REM ***  CSR  ***
5 REM ************
10 VS 5: CLS
20 CSR 9,12
30 PRINT "HELLO"
40 STOP
```

For further information on this command refer to chapter 8 where it is discussed at some length.

RELATED KEYWORDS: PRINT, INPUT

CTLSPR
SYNTAX : **CTLSPR p,x**

ABBREVIATION : **CT.**

Declares the global parameters for the sprite graphics commands.

The CTLSPR command is used to define the parameters of sprites in much the same way as LET is used to define a variable. The first parameter, p, is used to describe the following functions:

p	Function	range for x
0	Sprite speed	0-255
1	Distance	0-7
2	No. of active sprites	0-32
3	No. of orbitting sprites	0-32
4	Set sprite x as PLOT SPRITE	0-32
5	No. of moving sprites	0-32
6	Sprite size	0= 8*8 sprite, normal size 1= 8*8 sprite, double size 2= 16*16 sprite, normal size 3= 16*16 sprite, double size

The example below sets all active sprites to size 3, sets 3 active sprites, 2 of which are defined as orbiting

```
20 CTLSPR 2,3: CTLSPR 3,2: CTLSPR 6,3
```

For further information on this command refer to chapter 10 where it is explained at some length.

RELATED KEYWORDS: ADJSPR, GENPAT, MVSPR, SPRITE

DATA
SYNTAX : **DATA v,v,v,v...**
: **DATA v$,v$,v$...**
: **DATA v,v$,v,v$...**

ABBREVIATION : **D.**

Statement used to store data within the body of a program.

DATA statements are used in conjunction with READ and RESTORE to access string and numeric data as and when required by a program. The facility is simple to use and provides BASIC programmers with a means of storing large quantities of information which can be READ into variables or (more usually) arrays at an appropriate point in a program. The mechanics of DATA statements are best understood by an example. The following program could be coded in a variety of ways, but is a simple way of demonstrating the power of this valuable keyword:

```
1 REM ************
5 REM *** DATA  ***
10 REM************
20 DIM A$(5,8): LET Y=2
30 REM ** READ/PRINT LOOP **
40 FOR C=1 TO 4
```

```
50 READ A$(C)
60 CSR 10,Y
70 PRINT C;" ";A$(C)
80 LET Y=Y+2
90 NEXT C
100 CSR 10,14
110 PRINT "ENTER YOUR STATUS (1,2,3,4) "
120 LET S$=INKEY$: IF S$="" THEN  GOTO 120
130 LET S=VAL(S$)
140 IF S<1 OR S>4 THEN  GOTO 120
150 CLS : CSR 10,10
160 PRINT "SO YOU'RE A ";A$(S)
200 DATA MAN,WOMAN,CHILD,MACHINE
```

One of the wonders of DATA statements is that they can be positioned anywhere in a program. So whilst our example has the statements in line 200, they could equally well have appeared in line 10 or line 27. This said, it is standard practice to place DATA statements at the end of a program, but the point to note is that the MTX will access them regardless of their position. So how do DATA statements work?

Well, let's take a look at our example. Having DIMensioned the A$ array at the beginning of the program, we use the loop to READ values into each element of the array. With the first pass of the loop the first element of the array is established as A$(1). When the computer encounters the first READ statement it places a pointer against the first item of DATA in line 200, which is then assigned to A$(1). Thus the string "MAN" is stored in A$(1). With the next pass of the loop the array element becomes A$(2), the computer's pointer moves on to the next DATA item ("WOMAN"), and so on until the loop is complete. Thus:

A$(1) = "MAN"
A$(2) = "WOMAN"
A$(3) = "CHILD"
A$(4) = "MACHINE"

We can store both string and numeric data in DATA statements. You'll notice that we've used the loop counter (C) in line 70 to number our status options. These could have been included in our DATA statement. Test this out by making the following modifications to the example:

```
20 DIM A$(5,10): LET Y=2
70 PRINT A$(C)
160 PRINT " SO YOU'RE A ";MID$(A$(S),3,7)
200 DATA 1 MAN,2 WOMAN,3 CHILD,4 MACHINE
```

However, it should be noted that under these conditions the numeric data items are stored as strings, not numeric values. When using DATA statements it is important to ensure that the variable or array variable used in the READ statement is appropriate to programming requirements. Thus:

```
1 REM **********
5 REM *** DATA2 ***
```

```
10 REM**************
20 DIM N(3)
30 FOR A=1 TO 3
40 READ N(A)
50 NEXT A
60 PRINT N(1)+N(2)+N(3)
100 DATA 4,6,2
```

will PRINT out 12, since N(A) is a numeric array, and the calculations can be performed on the elements of the array. However:

```
1 REM **************
5 REM *** DATA3 ***
10 REM**************
20 DIM N$(3,2)
30 FOR A=1 TO 3
40 READ N$(A)
50 NEXT A
60 PRINT N$(1)+N$(2)+N$(3)
100 DATA 4,6,2
```

will simply return 462, since the N$ array is a string array, and hence line 60 performs a concatenation, not a calculation.

We have seen that the computer READs DATA, by placing a pointer against each item, and assigning it to the READ variable or array element. But what happens when the pointer reaches the last DATA item? Well, if the machine is asked to READ on more occasions than there are DATA items, processing will be halted with a "No data" error. You can test this out by increasing the value of the loop counter in line 30 of the last example program. So once the DATA statements have been READ, they can't to re-read. Well, not unless we make use of the RESTORE facility.

When a RESTORE statement is executed, the computer returns its DATA pointer to the first DATA item in a specified line. To see this in action, change the line 30 loop counter back to its original value (3) and add the following line:

```
70 GOTO 30
```

If you RUN the program all will be well until the MTX tries to start the READ loop for a second time. Since the DATA pointer reached the last data item the first time the READ loop was executed, there is nothing further to assign to the N$(A) array, and the program halts with a "No data" error. Now add the following line and RUN the program again:

```
65 RESTORE 100
```

This little addition will enable our program to RUN indefinitely. The RESTORE statement is telling the MTX to RESTORE the DATA pointer to the first DATA item in line 100.

The ability to RESTORE to a specific line number is a valuable feature of MTX BASIC. Many micros only permit DATA to be RESTOREd in its entirety. The option of using RESTORE to access and re-access specific sections of DATA adds considerable flexibility to the programming potential of DATA statements.

Finally. it is possible to READ different types of DATA in a single statement:

```
10 REM ****************
13 REM ***  DATA4  ***
15 REM ****************
20 DIM N$(3,4): DIM N(3)
30 FOR A=1 TO 3
40 READ N$(A),N(A)
50 NEXT A
60 PRINT N$(1)+N$(2)+N$(3)
70 PRINT N(1)+N(2)+N(3)
100 DATA MEM,2,0,3,TECH,5
```

Having hopefully established the importance of DATA statements, a word of warning about the manner in which the command itself must be entered. Unless the command is keyed-in in its abbreviated form the computer enters a leading space before the first DATA item, which is then READ into the first READ variable. You should remember this even when editing a DATA statement when you should wipe out DATA and insert D. before altering a line. This is actually very important since an extra character in a data item could stop the program if you have DIMensioned a string array leaving only just enough space for each data item.

Unlike many other micros, the MTX READs quotes ("") as DATA items and assigns them to the READ variable. All ASCII characters can be included in DATA statements as data items, with the exception of commas, which are interpreted as DATA separators.

If a program attempts to READ string DATA with a numeric variable, it will be halted with a 'Mismatch' error report.

RELATED KEYWORDS: READ,RESTORE,DIM,LET

DIM

SYNTAX : **DIM v(i,j,...)**
: **DIM v$(i,j,...)**

ABBREVIATION : **DI.**

DIM statements allow programmers to allocate memory space for the storage of string or numeric arrays.

BASIC offers two ways of storing data in variables. You can either use the standard format (LET A=34) or dimension an array. When the former method is employed, the value assigned can be re-defined, but the variable can only return a single value or string. However, by dimensioning an array you can assign a list of values to an array variable. We can create both string and numeric arrays, so let's kick off by taking a look at simple one dimensional numeric arrays.

NUMERIC ARRAYS

The need for dimensioning an array is quite simple. If we want to store a list of items, we have to let the

computer know how much space it has to reserve in memory for its storage. When an array is dimensioned, the variable name is always followed by a number enclosed in brackets. This value to tells the MTX the maximum number of items you want to assign to the array list. Thus:

```
10 DIM A(8)
```

allocates enough memory space for the storage of eight numbers. The first numeric value will be stored in A(1), the second in A(2), the third in A(3) and so on through to A(8). Key in the following program and we will then run through it to see what's happening.

```
10 REM ***************
13 REM ***  DIM 1  ***
15 REM ***************
20 DIM A(8)
30 FOR B=1 TO 8
40 LET A(B)=B*2
50 PRINT A(B)
60 NEXT B
```

This program will print the numbers 2, 4, 6, 8...16 to the screen. But, this could have done that without using an array we hear you cry. Well, this is very true, but using this format all of the values of the loop's variable B have been stored in the elements of the array, whereas a non-array variable would only retain the last value that it had been assigned. To demonstrate this try:

```
PRINT A(4)
```

or

```
PRINT A(2)
```

and 8 and 4 will be PRINTed to the screen. However, if we had keyed in:

```
1 REM ***************
3 REM ***  DIM 1  ***
5 REM ***************
10 FOR A=1 TO 8
20 LET B=A*2
30 PRINT B
40 NEXT A
```

after running the program, the multiplication variable B will always be equal to its last value, 16. Let's take a close look at our first example and see exactly what's going on.

In line 20 we DIMensioned our array, so the MTX is now ready for 8 elements of data. We then created a FOR...NEXT loop. In line 40 we have rather a curious construction. The reason for this is that although we have DIMensioned our array we have not actually told our silicon wonder what is going to be put into it. This line says, 'let the array A(B) be equal to the value of B*2'. The first time through the loop B will be equal to 1 so the first element of the array will be labelled A(1), which creates the statement.

A(1)=B*2

and as B is currently equal to 1, the value assigned to the array element A(1) is 2. The next time through the loop B is equal to 2, so the second element is labelled A(2) and assigned the value of 4. The process is repeated until the FOR...NEXT loop has run its course, and the final array element (A(8)), has been assigned a value (16).

DATA AND DIM

The value of this method of data storage becomes apparent when you realise that arrays enable us to access each of their elements simply and efficiently. Let's take a look at their use in conjunction with DATA statements, since this will demonstrate the ease of assignment and recall.

```
1 REM ***************
5 REM ***  DIM 2  ***
10 REM***************
20 DIM A(5)
30 FOR B=1 TO 5
40 READ A(B)
50 PRINT A(B)
60 NEXT B
70 FOR X=1 TO 5
80 PRINT A(X)
90 NEXT X
200 DATA 1,65,98,0,90
```

By using an array in conjunction with the READ and DATA statements it makes it possible to store all the DATA in 'different' variable names. In our example the first loop assigns and PRINTs the values of the A(B) array, while the second loop shows how each element can be recalled.

So far we have only looked at numerical arrays. Let's move on and see how string arrays can be assigned. This is achieved in exactly the same way, except you have to use the dollar sign in order to indicate that it is a string array that you want to DIMension. Thus:

DIM A$(8)

will set aside enough space in the memory for eight string characters.

STRING ARRAYS

There is one major difference between string and numeric arrays. Although the above command will create enough space for 8 items of string data, it will only leave space for one character per item. In order to set enough space side for multi-character string items, you have to assess the length of the longest string that you will want to assign to an array. Let's say that the longest string required is 10 characters long:

DIM A$(8,10)

This tells the MTX to set aside enough room for eight pieces of string data of ten characters long. This is called a multi-dimensional array, since more than one value is used to DIMension it. (A multi DIMensional array can have multiple dimensioning parameters, for example, DIM A$ (8,4,7,9).) If this second number is

ommitted and you try to assign a string to one of its elements that is longer than a single character a 'No space' message will be returned.

```
1 REM **********************
5 REM *** STRING ARRAYS ***
10 REM**********************
20 DIM A$(5,9)
30 FOR B=1 TO 5
40 READ A$(B)
50 PRINT A$(B)
60 NEXT B
100 DATA MONDAY,TUESDAY
110 DATA WEDNESDAY,THURSDAY
120 DATA FRIDAY
```

The DIM statement in line 20 ensures that there is enough space in memory for five data items of nine characters in length (the longest item of data is WEDNESDAY, which is nine characters long). The FOR...NEXT loop makes the computer READ the DATA items and store their contents into the array variable (A$(B)), where B will be 1 the first time through the loop, 2 the second time through and finally 5. Thus A$(1) will hold the string MONDAY and A$(5) will hold the string FRIDAY. Notice that it's only necessary to include a string array's subscript when recalling a string value. The second parameter can be omitted. The elements of the A$ array are PRINTed out in the loop.

RELATED KEYWORDS: LET

DRAW
SYNTAX : DRAW x

ABBREVIATION : DR.

The DRAW command draws a line of length x in the current 'plotting direction'. The plotting position is updated when DRAW is completed. The example below draws a horizontal line from the centre of the screen, followed by a similar vertical line:

```
1 REM **************
3 REM ***  DRAW  ***
5 REM **************
10 VS 4: CLS
20 ANGLE 0
30 PLOT 127,96
40 DRAW 50
50 PHI PI/2
60 DRAW 50
70 GOTO 70
```

For further information on this command refer to chapter 9 where it it discussed at some length.

RELATED KEYWORDS: ANGLE, ARC, PHI, PI, PLOT

DSI
SYNTAX : DSI

ABBREVIATION : DS.

Enables the use of all of the keyboard keys where the result of any key-press is displayed directly on the screen.

DSI is short for Direct Screen Input. DSI remains activated until the RET key is presed. When using the DSI command, the following CTL and ESC sequences can also be entered:

SEQUENCE		EFFECT
CTL	W	Tab back
CTL	]	Set page mode
CTL	\	Set scroll mode
CTL	^	flashing cursor ON
CTL	_	flashing cursor OFF
CTL	D	then A to O PAPER colour A (1) to O (15)
CTL	F	then A to O INK colour a (1) to O (15)
ESC	I	Insert a line
ESC	J	Delete a line
ESC	K	Duplicate a line

Our example program allows you to roam freely about the screen. Try using the cursor keys, changing colours etc. Note that the break key is not operational during DSI, but instead generates a CTL C.

```
1 REM **************
3 REM ***  DSI   ***
5 REM **************
10 VS 4: CLS
20 DSI
30 STOP
```

RELATED KEYWORDS: CRVS, VS

EDIT
SYNTAX : EDIT In

ABBREVIATION : E.

Extracts the program line specified by the command's argument (In), and makes it available for EDITing.

EDIT provides access to the MTX's powerful EDITing facilities. When developing a program it is often necessary to make alterations to statements that have already been entered. Rather than type in the entire line again, we use the EDIT facility. Suppose we want to alter line 60. By entering the following as a direct command: EDIT 60 or E.60 we cause line 60 to appear on the MTX's EDIT screen (at the bottom of the VDU). If the line specified doesn't exist, the machine will not throw up an error message, but the EDIT screen will remain blank and the computer will simply await further instructions.

Once the program line has appeared on the EDIT screen, programmer's have full access to the MTX's extensive editing facilities. For further details see chapter 2.

Note that you can only call down an entire program line for editing. So if you want to access a statement within a multi-statement line, you must EDIT the entire line. For obvious reasons, EDIT is almost always used as a direct command. However, it can be used within a program, as the following example from our Sound chapter demonstrates:

```
10 REM **************
13 REM ***  EDIT  ***
15 REM **************
20 SBUF 2
30 SOUND 0,3906,900,1,-1,640,1
40 EDIT 30
```

Line 40 presents us with line 30 to EDIT, whilst the SOUND created by the statement drones on.

For ASSEMBLER editing see chapter 12.

RELATED KEYWORDS: none

EDITOR
SYNTAX : **EDITOR <variable list>**

ABBREVIATION : **EDITO.**

Gives you the ability to accept input from a defined area of the screen.

EDITOR always uses virtual screen 0 and the size of this screen can be set using CRVS. When exiting EDITOR the screen will remain set to screen 0 and must be changed by the programmer using VS.

The following example uses EDITOR to store the inputted data in the string variable A$. The Virtual Screen 0 is defined to be a single line deep and 5 characters wide:

```
1 REM ****************
3 REM ***  EDITOR  ***
5 REM ****************
10 VS 5: CLS : PAUSE 200
20 CSR 5,10: PRINT "ENTER NO MORE THEN 5 LETTERS"
30 CRVS 0,0,5,11,5,1,40
40 EDITOR A$
50 VS 5
60 CSR 5,15: PRINT A$
70 GOTO 10
```

RELATED KEYWORDS: CRVS, VS

ELSE
SYNTAX : **IF c THEN s ELSE s**

ABBREVIATION : **EL.**

Optional part of the IF...THEN conditional test, which facilitates the inclusion of a False option.

The ELSE facility in MTX BASIC adds flexibility to the IF...THEN construction. With a statement such as:

```
50 IF C=-1 THEN  PRINT "TRUE"
```

the PRINT statement will only be executed if C is equal to -1. In fact the computer won't even bother to look at the rest of the statement if the initial condition is False. However, by including ELSE in the statement we can provide alternative instructions to be implemented as the False option:

```
50 IF C=-1 THEN  PRINT "TRUE" ELSE  PRINT "FALSE"
```

The value of such a facility is clear. The construction enables programmers to produce clear and economic code, and avoids such clumsy duplications of statements as:

```
50 IF C=-1 THEN  PRINT "TRUE"
60 IF C<>-1 THEN  PRINT "FALSE"
```

The example program has been designed to show ELSE is action in a variety of formats.

```
10 REM ****************************
15 REM
20 REM ***    ELSE  EXAMPLE    ***
25 REM
30 REM ****************************
40 DIM B$(4,4),B(4),V$(4,4),X(4)

60 LET F=1: CSR 8,9
70 PRINT "A=B AND X>Y "
80 REM ****************************
85 REM
90 REM ***   DATA/PRINT LOOP   ***
95 REM
100 REM ***************************
110 FOR A=1 TO 4
120 GOSUB 300
130 READ X(A),V$(A)
140 PRINT V$(A)
150 GOSUB 600
160 CSR X(A),9: PRINT B$(A)
165 PAUSE 200
170 NEXT A
180 REM ***************************
185 REM
190 REM ***  DISPLAY  RESULTS  ***
195 REM
200 REM ***************************
```

```
210 LET F=2: GOSUB 300
220 LET R=(B(1)=B(2) AND B(3)>B(4))
230 IF R=-1 THEN  PRINT "TRUE " ELSE  PRINT "FALSE "
240 STOP
245 REM ***************************
250 REM
255 REM ***   END OF PROGRAM   ***
260 REM
265 REM ***************************
270 REM
290 REM
295 REM
300 REM ***************************
305 REM
310 REM *** DISPLAY SUBROUTINE ***
315 REM
320 REM ***************************
330 CSR 1,7
340 IF F=1 THEN  GOSUB 400 ELSE  GOSUB 500
350 RETURN
400 REM ***************************
405 REM
410 REM ***  TRUE  SUBROUTINE  ***
415 REM
420 REM ***************************
430 CSR 1,7: PRINT "EXAMPLE: "
440 CSR 5,12: PRINT "ENTER VALUE FOR ";
450 RETURN
500 REM ***************************
505 REM
510 REM ***  FALSE SUBROUTINE  ***
515 REM
520 REM ***************************
530 CSR 1,7: PRINT "THE EXPRESSION: "
540 CSR 5,12: PRINT " IS LOGICALLY ";
550 RETURN
600 REM ***************************
605 REM
610 REM ***        INPUT         ***
615 REM
620 REM ***************************
630 LET B$(A)=INKEY$
640 IF INKEY$="" THEN  GOTO 630
650 LET T=ASC(B$(A))
660 IF T<48 OR T>57 THEN  GOTO 630
670 LET B(A)=VAL(B$(A))
680 RETURN
700 DATA  8,A,10,B,16,X,18,Y
```

RELATED KEYWORDS: IF, THEN

EXP
SYNTAX : **v=EXP(n)**

ABBREVIATION : **EX.**

This function raises e to the power of its argument (n).

As you probably know, if we talk of raising 3 to the power of 4 (3^4), we are calculating:

3*3*3*3

and the EXP function calculates the value of 2.71828183 raised to the power of the value contained in brackets (n). The value of e is a magic number well known to mathemeticians, not least because:

EXP(n)

returns the natural antilog of n. The example program demonstrates EXP along with a number of other MTX functions in the evaluation of complex mathematical expressions:

```
5 REM  ****************
10 REM ***   EXP    ***
15 REM ****************
20 CLS
30 INPUT "PLEASE ENTER TWO NUMBERS SEPERATED BY A COMMA ";X,Y
40 IF LN(X)<0 THEN  LET X=1/X
50 PRINT SQR(X)*EXP(Y)
60 INPUT "ANOTHER GO? (Y/N) ";N$
70 IF N$="Y" THEN  GOTO 10 ELSE  STOP
```

RELATED KEYWORDS: LN

FOR...TO...(STEP)...NEXT

SYNTAX : FOR v=n TO n (STEP n) NEXT v
FOR :
TO :
STEP :
NEXT :

ABBREVIATION
FOR : FO.
TO : none
STEP : STE.
NEXT : N.

A statement which creates a loop structure that processes the instructions contained within the body of a loop (the amount of times specified in the opening line). The loop is terminated by a NEXT statement.

In the final analysis, the most valuable quality of any microcomputer lies in its ability to endlessly carry out boring, repititious tasks without complaint. BASIC offers a number of structures which enable the heartless programmer to force his/her micro to carry out such tasks effeciently, and the FOR...NEXT loop is one of the most important.

Loops are easy to use and, once mastered, provide the programmer with one of the most valuable facilities of the BASIC language. Take a simple example. Say we want to PRINT the alphabet. Without a loop the coding would be hideously tedious, requiring an individual PRINT statement for each letter. The FOR...NEXT construction makes life considerably easier:

```
10 REM **********************
13 REM ***  FOR...NEXT  ***
15 REM **********************
20 FOR A=0 TO 25
30 PRINT CHR$(A+65),
40 NEXT A
```

What could be simpler? Let's take it line by line. In line 20 the number of times that the instructions within the loop are to be executed is established. In this case it is twenty-six times (0-25). Thus the first time the loop is processed A (the loop counter) will be equal to 0, the second pass of the loop A=1, the third A=2, and so on until A=25. In line 30 the value of the loop counter (A) is included in the argument of the CHR$ statement, where it is added to 65 (the ASCII code for the letter "A"). When it reaches line 40 the NEXT statement checks the value of the loop counter to see if it has reached its maximum value (25 in this case). If A is not equal to 25, the NEXT statement forces the computer to return to the beginning of the loop and start the process all over again.

In the example above the value of A is incremented in steps of 1. This will always be the case, unless we make use of the optional STEP command. Key in the following modification to our example:

```
20 FOR A=0 TO 25 STEP 2
```

If we RUN the program again, instead of PRINTing out A,B,C,...Z, the computer will increment the value of A in STEPs of 2, thus PRINTing every other letter of the alphabet (i.e. thirteen passes of the loop).It is also possible for the STEP size to be negative. For example:

```
10 REM **********************
13 REM *** FOR...NEXT 2 ***
15 REM **********************
20 FOR A=25 TO 5 STEP -5
30 PRINT A
40 NEXT A
```

The possibilities are virtually endless! However, it should be noted that when a negative STEP size is specified the end value of the loop counter will be the specified end value plus the step size. For instance in the above example the end value will be 10 not 5. There's no reason why the STEP size should be a whole number. Try this:

```
20 FOR A=10 TO 5 STEP -.25
```

In fact, the FOR...NEXT loop is so flexible that all its values can be non-integer, variables or even calculated expressions. Add these lines to test these wild claims:

```
1Ø LET X=56: LET C=PI/2
2Ø FOR A=2*4 TO X/3 STEP C
```

If you use a NEXT statement without a corresponding FOR statement the program will halt with a NO FOR error message.

The inclusion of the loop variable in the NEXT statement is optional, but it is advisable to leave it in since it makes a program much easier to read, particularly if you have created a series of loops nested within each other.

Finally, note that you should never jump out of a loop with a GOTO or a GOSUB statement. If a crisis forces you into a position where a loop jump is unavoidable you must always return to the loop and allow it to complete its specified number of passes. Failure to comply with this rule will sooner or later result in the

program crashing.

RELATED KEYWORDS: none

GENPAT
SYNTAX : GENPAT p,n,d1,d2,d3,d4,d5,d6,d7,d8

ABBREVIATION : GE.

Enables the GENeration of PATterns. This command is used to generate the PATterns for both user-defined characters and sprites.

The last eight of the statement's parameters (d1 to d8) are the items of data that define the shape to be generated. The parameter p defines the type of shape to be created whilst n determines the actual character or sprite shape.

p	meaning	range of n
0	ASCII characters	32 to 127
1	User-defined characters	129 to 154
2	Multi-colour characters	147 to 154
3	sprite 8*8 pattern	1 to 127
4	sprite 16*16 pattern top left corner	1 to 31
5	sprite 16*16 pattern bottom left corner	as above
6	sprite 16*16 pattern top right corner	" "
7	sprite 16*16 pattern bottom right corner	" "

The example program below creates a user-defined character to replace the '*' shape. After this program has been RUN, whenever the '*' key is pressed, a user-defined skull shape will be PRINTed to the screen instead of an asterisk. In this example we are using the multiplication sign, and you should note that the computer continues to act upon the multiply symbol, even when it PRINTs a skull shape. For instance, 3 <skull> 2=6. So only the shape of a particular character has been changed, not its effect.

```
1 REM **********************
5 REM ***   GENPAT      ***
9 REM **********************
10 VS 4: CLS
20 GENPAT 0,42,60,126,219,255,231,126,36,60
30 CSR 10,12: PRINT "* * * * * *"
40 GOTO 40
```

Changing the parameter p and the number n allows you to define non ASCII characters or sprites.

When using user-defined characters the character set remains redefined at the end of the program. To return to the standard character set type <ESC> followed by B and then 0.

For further information on this command refer to chapters 8 and 10.

RELATED KEYWORDS: ADJSPR, CHR$, PRINT, MVSPR, SPRITE, CTLSPR

GOSUB

SYNTAX : **GOSUB** ln
: **ON** v **GOSUB** ln

ABBREVIATION : **GOS.**

A statement which transfers control from the main body of a program to a subroutine starting at line number ln. Control is returned to the main program by the means of the compulsory RETURN statement.

The creation of subroutines utilising GOSUB...RETURN statements provide BASIC programmers with a means of creating code which is not only processed efficiently, but is also easy to follow. A GOSUB statement resembles GOTO in as much as it passes processing control to the specified line number (rather than following the line numbers in sequence). However, GOSUB is considerably more powerful than GOTO because as well as directing control to a specified module it also stores a return address in memory. Thus once the instructions contained in the subroutine have been executed, a RETURN statement forces control back to the program line following the original GOSUB statement.

The line number specified by a GOSUB statement must be literally quoted. Variables or the results of calculated expressions cannot be entered.

One important rule to remember when using subroutines is that they should never be jumped out of with a GOTO, but should always be allowed to RETURN to the main body of the program. Apart from being the height of bad programming practice, jumping out of subroutines and not RETURNing will soon corrupt the MTX's memory stack and crash the program. This said, it is quite acceptable to create subroutines that call other subroutines (such structures are known as nested subroutines), but care must be taken to ensure that ultimately control is systematically RETURNed to the statement that follows the original GOSUB statement.

As well as promoting the creation of clear and efficient code, GOSUBs are obviously an important means of saving memory as they enable a single section of code to be executed any number of times.

Before moving on to our example program, one final word of warning. Since it is standard practice to place subroutines at the end of a program it is important to make sure that when the main body of the program has been processed it is halted before the computer reaches the subroutines (which must only be entered or called with a GOSUB statement).

```
5 REM  **********************
10 REM ***      GOSUB      ***
15 REM **********************
20 REM
80 REM **********************
90 REM ***      INPUT      ***
95 REM **********************
100 REM
110 CLS : INPUT "KEY IN A RADIUS ";R
120 REM
130 REM ************************
135 REM ***  CIRCUMFERENCE  ***
140 REM ************************
150 LET C=2*PI*R
160 LET Z=C
170 GOSUB 300
180 PRINT "CIRCUMFERENCE IS ";Z
190 REM **********************
195 REM ***       AREA       ***
200 REM **********************
210 LET A=PI*R^2
220 LET Z=A
230 GOSUB 300
```

```
240 PRINT "AREA IS ";Z
250 STOP
260 REM **********************
265 REM *** END OF PROGRAM ***
270 REM **********************
300 REM **********************
305 REM ***   SUBROUTINE    ***
310 REM **********************
320 LET Z=INT(100*(Z+.005))
330 LET Z=Z/100
340 RETURN
350 REM ************************
355 REM *** END OF SUBROUTINE **
360 REM ************************
```

Finally, multiple branching in a program can be coded by coupling GOSUB with an ON statement. This structure is commonly used to direct control in menu-driven programs, where user INPUT determines the selection of a subroutine.

```
5 REM  **********************
10 REM ***    ON GOSUB    ***
15 REM **********************
20 REM
25 REM **********************
30 REM ***  MAIN PROGRAM  ***
35 REM *** SCREEN DISPLAY ***
40 REM **********************
50 CLS : CSR 2,1: PRINT "THIS  PROGRAM CALCULATES THE AREA"
60 CSR 2,3: PRINT "OF A CIRCLE, OR AN ANGLE'S COSINE"
70 CSR 2,5: PRINT "(MEASURED  IN RADIANS), FROM  THE"
80 CSR 2,7: PRINT "VALUE OF THE INPUT"
90 PAUSE 5000
100 CLS
105 REM **********************
110 REM ***  VALUE INPUT  ***
115 REM **********************
120 INPUT "ENTER RADIUS OR ANGLE ";V: CLS
125 REM **********************
130 REM *** ON GOSUB CHOICE**
135 REM **********************
140 CSR 3,1: PRINT "ENTER NUMBER OF CALCULATION"
150 CSR 3,3: PRINT "REQUIRED (0 OR 1) "
160 LET A$="THE AREA OF A CIRCLE "
170 LET B$="THE COSINE OF AN ANGLE"
180 CSR 3,7: PRINT "0. ";A$
190 CSR 3,9: PRINT "1. ";B$
195 REM **********************
200 REM *** ON GOSUB INPUT **
205 REM **********************
210 INPUT C
230 ON C GOSUB 300,400
235 REM **********************
240 REM ***     AGAIN     ***
245 REM **********************
250 CSR 3,13:INPUT "ANOTHER VALUE (Y/N)?";S$
270 IF S$="Y" THEN  GOTO 100 ELSE  STOP
```

```
300 REM **********************
305 REM ***AREA SUBROUTINE***
310 REM **********************
320 LET A=PI*V^2: CLS
330 CSR 3,3: PRINT A$
340 CSR 3,5: PRINT "WITH A RADIUS OF ";V
350 CSR 3,7: PRINT "IS ";A
360 RETURN
370 REM **********************
375 REM ***END OF AREA SUB***
380 REM **********************
390 REM
400 REM **********************
410 REM **********************
415 REM ***COS  SUBROUTINE***
420 LET B=COS(V): CLS
430 CSR 3,3: PRINT B$
440 CSR 3,5: PRINT "WITH A RADIAN OF ";V
450 CSR 3,7: PRINT "IS ";B
460 RETURN
470 REM **********************
475 REM *** END OF COS SUB ***
480 REM **********************
```

In our example program (which makes extensive use of REM statements to clarify the structure of the program), the ON...GOSUB statement is used to determine whether the V INPUT is processed by subroutine 0 or 1. If the value of INPUT C is 0 the program will GOSUB 300 and use V to calculate the area of a circle. If C=1 GOSUB 400 will be called upon to calculate the COSine of the value entered.

RELATED KEYWORDS: GOTO, ON, RETURN

GOTO

SYNTAX : **GOTO ln**
: **ON v GOTO ln**

ABBREVIATION : **G.**

Statement which passes control to the line number specified by ln which must be literally quoted.

As you probably know, when a program is RUN the computer will execute statements according to the order specified by the program's line numbers, unless this process is disrupted by either a GOTO, GOSUB, RETURN, ON GOTO, or ON GOSUB statement. The only other facilities in MTX BASIC capable of diverting a program's linear flow are FOR...NEXT loops and the ASSEM statement (which passes control to a machine code subroutine).

All of these statements are important in different ways and each enhances the programmer's capacity to code the flow of a program to suit the needs of a particular problem. As soon as you start writing your own programs it will immediately become clear why linear processing by line numbers would simply not be practical.

A GOTO statement is one of the simplest means of re-directing control, and it performs in much the same way as a GOSUB. The major difference between the two statements is that instead of passing control to a subroutine and RETURNing to the main program, GOTO simply jumps to the specified line number and continues processing in the normal way. No return address is stored when a GOTO statement is executed and hence there is no way that control can be returned to the code between the statement and line it specifies (except by using another GOTO).

One of the numerous and relentlessly flaunted Golden Rules of Programming decrees that GOTOs should only be used to pass control forward in a program A swift glance through the programs in the present volume will give you some idea of just how rigorously this rule is observed. This said, the principle behind this particular sacred cow is both sound and sensible. Like all statements which manipulate the control of a program, GOTO is a powerful command which should be used with care and in moderation. The advocates of the GOTO Golden Rule are essentially over-reacting to the rather depressing fact that more often than not GOTO statements are used by desperate programmers attempting to 'patch' their flawed and poorly structured creations. This, coupled with the undeniable observation that one GOTO invariably leads to another, has prompted BASIC purists to associate GOTO statements with the spectre of 'spaghetti' programming. (A spaghetti program is one whose flow of control is as easy to unravel as a bowl of the aforementioned pasta!)

As a general rule, if you find you are developing a program that seems to require a large number of GOTOs, it is almost certain that the most efficient solution to the problem you are tackling has eluded you. In other words, think again!

Having devoted a great deal of space to the pros and cons of using GOTO statements, let's take a look at an example which illustrates some of the points we have discussed:

```
10 REM ****************
13 REM ***  GOTO 1  ***
15 REM ****************
20 CLS
30 INPUT "ENTER A VALUE ";V
40 IF V>10 THEN  PRINT "YOUR  NUMBER WAS GREATER T
HAN 10": GOTO 60
50 PRINT "YOUR NUMBER WAS LESS THAN ELEVEN"
60 PRINT " ANOTHER VALUE (Y/N)?"
70 IF INKEY$="" THEN  GOTO 70
80 IF INKEY$="Y" THEN  GOTO 20
90 STOP
```

The inclusion of the statement in the lines 70 and 80 is efficient, acceptable and clear. However the first use of GOTO in line 40 could have been more coherently coded by the inclusion of the ELSE statement. Thus:

```
10 REM ****************
13 REM ***  GOTO 2  ***
15 REM ****************
20 CLS
30 INPUT "ENTER A VALUE ";V
40 IF V>10 THEN  PRINT "YOUR  NUMBER WAS GREATER T
HAN 10" ELSE  PRINT "YOUR NUMBER WAS LESS THAN ELE
VEN"
50 PRINT " ANOTHER VALUE (Y/N)?"
60 IF INKEY$="" THEN  GOTO 60
70 IF INKEY$="Y" THEN  GOTO 20 ELSE  STOP
```

RELATED KEYWORDS : GOSUB, ON, RETURN

GR$
SYNTAX : **a$=GR$ x,y,b**

ABBREVIATION : **GR.**

Reads a bit pattern from the graphics screen, returning the value as a character.

The GR$ function is particularly useful if you want to PRINT the graphics screen to a high-resolution printer. However, before the PRINT operation can be executed, GR$ must be assigned to a string variable (see syntax example). Another use for GR$ is to detect the presence or otherwise of plotted points on the graphic screen.

The values x and y are the co-ordinates, in pixels (256*192) on the full or virtual screen. The b parameter is the number of bits to be read. The bits are read vertically downwards, starting at x,y. For instance:

```
LET A$=GRS(30,100,5)
LPRINT A$
```

gives a character made up as follows:

BIT 7 0
BIT 6 0
BIT 5 0
BIT 4 PIXEL AT 30,100
BIT 3 PIXEL AT 30,99
BIT 2 PIXEL AT 30,98
BIT 1 PIXEL AT 30,97
BIT 0 PIXEL AT 30,96

If all of the above pixels are ON then the result of GR$ (30,100,5) would be (16+8+4+2+1=0)

The value of GR$ when creating hard-copy screen dumps is that user-defined characters on printers such as the MEMOTECH DMX-80, and others, are defined in terms of columns rather than the horizontal rows used by micros (see Chapter 8 for full details of user-defined characters).

RELATED KEYWORDS: LINE, PLOT

IF...THEN...(ELSE)
SYNTAX : **IF c THEN statement (True) (ELSE statement (False))**
BASIC Token **IF : THEN**
ABBREVIATION **IF : none**
THEN : T.

Statement which enables the MTX to test conditions and, contingent upon their outcome, execute or ignore specific instructions.

The IF...THEN...ELSE statement is what is known as a decision structure. The condition (c) following IF can be any legal BASIC expression, using variables, logical operators, strings or numbers. The statement that follows the THEN part of the decision structure can be any legal BASIC instruction. The ELSE is enclosed in brackets above (although not when it is actually used in a program) because it is an optional element in this format. The statement that follows ELSE can be any legal BASIC instruction.

When the construction is used without ELSE, if the condition(s) specified in the first part of the statement are satisfied (True), the instructions which follow THEN are executed. If the specified condition(s) are False (not satisfied) control passes to the next line in the program. If ELSE is included in the statement, the

instructions following ELSE constitute a False option which the computer executes if the conditional expression is not satisfied. Multi-statement lines should be avoided when this construction is used as they tend to be very confusing.

```
5 REM  ************************
1Ø REM
15 REM ***     IF....THEN     ***
2Ø REM
25 REM ************************
3Ø INPUT "ENTER YOUR SURNAME ";N$
4Ø INPUT "ENTER YOUR SEX (M/F) ";S$
5Ø CLS
6Ø IF S$="M" THEN  LET S$="MR. ": GOTO 8Ø
7Ø LET S$="MS. "
8Ø PRINT "YOUR NAME IS ";S$;N$
```

In the above program, MR will be PRINTed if your INPUT (S$) is an "M". If M is not entered the variable S$ will be assigned MS which is PRINTed in line 80. This program could have been more efficiently coded with the inclusion of the ELSE statement. This modification does away with the need for the GOTO in line 60 and generally clarifies the program.

```
5 REM  ************************
1Ø REM
15 REM ***  IF...THEN...ELSE  ***
2Ø REM
25 REM ************************
3Ø INPUT "ENTER YOUR SURNAME ";N$
4Ø INPUT "ENTER YOUR SEX (M/F) ";S$
5Ø CLS
6Ø IF S$="M" THEN  LET S$="MR. " ELSE  LET S$="MS. "
7Ø PRINT "YOUR NAME IS ";S$;N$
```

RELATED KEYWORDS: AND, OR, NOT, GOTO, GOSUB

INK
SYNTAX : **INK i**

ABBREVIATION : **I.**

Sets a foreground colour which is determined by the value specified in the statement's argument (i).

INK statements can be used in both the graphics and text modes, and allow us to specify the foreground colour of the screen.

The following colour table lists the wide range of colours available on the MTX, along with the codes which are used to access them in INK (and PAPER) statements:

0 TRANSPARENT
1 BLACK
2 MEDIUM GREEN
3 LIGHT GREEN
4 DARK BLUE
5 LIGHT BLUE
6 DARK RED

7	CYAN
8	MEDIUM RED
9	DARK YELLOW
10	LIGHT YELLOW
11	DARK GREEN
13	MAGENTA
14	GREY
15	WHITE

Thus:

```
10 INK 4
20 PRINT "COLOUR"
30 GOTO 30
```

will print COLOUR in dark blue on the screen. When we use INK in the text mode, the command alters the foreground colour of the entire screen. For example, if we use INK 6 (dark red) in a statement, as well as all subsequent PRINT items, all items currently on display will now appear in red, regardless of what INK colour was in action when they were printed.

When INK is used in the graphics mode (VS 4), the statement sets the INK colour for all subsequent PRINT items, leaving those already on display unaffected. To appreciate the difference between the two modes, enter the following example and RUN it:

```
5 REM  ************
10 REM ***  INK  ***
15 REM ************
20 PAPER 1
30 FOR A=1 TO 14
40 INK A
50 FOR B=1 TO 6
60 PRINT "COLOUR",
70 NEXT B
80 PAUSE 1000
90 NEXT A
100 CLS
110 GOTO 30
```

BReaK into the program when you've had enough. By now the effect of INK on the text screen should be clear. Now add the following line to the example to see INK on the graphics screen:

```
17 VS 4: CLS
```

RELATED KEYWORDS: PAPER, VS, PRINT

INKEY$
SYNTAX : A$=INKEY$
ABBREVIATION : INKE.

Function which scans keyboard to see if a key is being pressed and, if it is, stores the value of that key in the string variable specified (A$).

INPUT halts a program and will not allow processing to continue until an appropriate form of data has been keyed in and the <RET> key pressed. There are occasions when a single character input is required and it is inconvienient to make the user of a program press <RET> before processing continues, INKEY$ offers a solution to this problem.

It should be stressed that by itself an INKEY$ statement does not halt a program. When the computer encounters such a statement it simply checks to see if a key is being pressed. If no key is being pressed, the program moves on. If the user is holding down a key, its value is stored in the statement's variable. So in its simple form INKEY$ isn't much use. However, if we take a look at one of the commonest roles for this command we will see how it can be used to halt a program until an appropriate response is obtained from the outside world.

When introducing a program it's often necessary to display screenfuls of data or instructions. Before moving from one screen of information to the next, we have to ensure that the user has had time to digest the data. The usual way of handling this is to ask the user to "PRESS ANY KEY TO CONTINUE". This is where INKEY$ comes into its own. Take a look at the following short example:

```
1 REM ***************
5 REM *** INKEY$  ***
7 REM ***************
20 CSR 14,10
30 PRINT "SCREEN1"
40 CSR 6,12
50 PRINT "PRESS ANY KEY TO CONTINUE"
60 LET A$=INKEY$
70 IF A$="" THEN  GOTO 60
80 CLS : CSR 14,10
90 PRINT "SCREEN2"
```

As you can see, it is line 70 that actually stops the program. It checks to see if A$ is a null string ("") - which it will be if no key is pressed - and continually returns to the INKEY$ statement until something is entered.

INKEY$'s value as an INPUT substitute is limited by the fact that it can only be used to receive single character string data. Thus if INKEY$ is used to input numeric data (only integer values between 0-9 remember), it must be converted into a numeric value using the VAL function.

Another situation in which INKEY$ often appears is when the keyboard is used to control simulated movement on the screen. The example program below demonstrates this role:

```
1 REM ***************
5 REM *** INKEY$  ***
10 REM***************
20 DIM B$(12)
30 LET X=2: LET Y=1
40 GOSUB 300
50 CSR X,10: PRINT "*"
```

```
60 GOSUB 200
70 CSR X,10: PRINT " "
80 IF A=80 THEN  LET X=X+1 ELSE  LET X=X-1
90 GOTO 50
180 REM ************************
185 REM *** DETECT KEY PRESS ***
190 REM ************************
200 LET A$=INKEY$: IF A$="" THEN  GOTO 200
210 LET A=ASC(A$)
220 IF A<79 OR A>81 THEN  GOTO 200
230 IF A=79 THEN  STOP
240 RETURN
280 REM ********************
285 REM ***  READS DATA  ***
290 REM ********************
300 FOR C=1 TO 3
310 READ B$: CSR 5,Y
320 PRINT "PRESS ";B$: LET Y=Y+1
330 NEXT C
340 RETURN
400 DATA O FOR END,P FOR RIGHT
410 DATA Q FOR LEFT
```

RELATED KEYWORDS: INPUT,IF,THEN

INPUT

SYNTAX : **INPUT v,v$**
: **INPUT "prompt";v,v$**

ABBREVIATION : **INP.**

A statement which allows the user of a program to enter string and numeric data which is then held in the INPUT variable(s).

INPUT statements are the most popular method of entering data for subsequent processing in a program. When the MTX reaches an INPUT statement the program will stop until the user keys in the appropriate form of data (string or numeric) and presses the <RET> key. The type of data it will accept is determined by the statement's format. For example:

```
10 INPUT n
```

will stop the program and PRINT a prompt (in the form of a question mark) on the screen. Since n is a numeric variable, the computer will only accept a numeric INPUT which, as the operator types it in, will be printed to the screen alongside the prompt. The program will continue processing from the line following the INPUT statement as soon as <RET> is pressed. If string data is entered (when numeric was expected) a question mark will appear after the characters entered and another prompt printed on the following line. The user must type in the appropriate type of data before the program can continue. When the MTX is satisfied

with the INPUT it assigns its value to whatever INPUT variable has been used in the statement (in this case n).

String data can be INPUT in much the same way. For example:

10 INPUT A$

will stop the program until characters have been entered and <RET> pressed. This format will accept any keyboard data (except a comma). However, any numbers that are entered will be stored as strings and can only be operated upon as such unless VAL is used.

Since the ? prompt gives the user of the program no indication as to the form of INPUT that is acceptable to the computer. INPUT statements are normally accompanied by an additional prompt which indicates the kind of response required by the program. A customised prompt can be coded in one of two ways:

```
1 REM ****************
3 REM *** INPUT 1 ***
5 REM ****************
10 PRINT "ENTER A NUMBER (1-19)"
20 INPUT N
```

or

```
1 REM ****************
3 REM *** INPUT 2 ***
5 REM ****************
10 INPUT "ENTER A NUMBER (1-19)";N
```

INPUT statements can contain more than one variable. For example:

```
10 INPUT A,A$,NAME$
```

is a perfectly acceptable construction. Processing will restart when the appropriate data has been entered for each of the INPUT variables, When using a single statement for multiple INPUTs, each data item entered should be separated by a comma. When using INPUT statements to obtain multiple entries of both string and numeric data it is obviously essential to present explicit prompts if incorrect entries are to be avoided. For example:

```
1 REM ****************
3 REM *** INPUT 4 ***
5 REM ****************
10 REM****************
20 PRINT "ENTER YOUR SURNAME FIRST"
30 PRINT "THEN YOUR AGE"
40 PRINT "AND FINALLY YOUR PHONE NUMBER"
50 PRINT "PRESS <RET> AFTER EACH ENTRY"
```

```
60 INPUT NAME$
70 INPUT AGE
80 INPUT NUM$
90 PRINT NAME$,AGE,NUM$
```

If this program is RUN the question mark prompt appears on the screen and the program stops until a string has been keyed in and <RET> pressed. Note that although the final INPUT in line 80 is a number it has been assigned to string rather than a numeric variable. This is to ensure that however the data is presented by the user (636 2101, 01- 636 2101, Beaminster 7463 or 6462192) it will be accepted as valid.

INPUT can only be used within a BASIC program, it will not be accepted as a direct command.

RELATED KEYWORDS: PRINT, INKEY$, CSR

INT
SYNTAX : v=INT(n)

ABBREVIATION : none

Function which returns the integer value of its argument (n). INT is used when it is necessary to ensure that a value is an integer (a whole number) rather than a floating point value. It is important to remember that the value returned by INT will be less than or equal to any positive number to which the function has been applied. This is because the action of INT rounds (n) down to the next lowest integer. For example:

```
5 REM **************
10 REM***  INT  ***
15 REM**************
20 LET I=INT(49.991)
30 PRINT I
```

will PRINT out 49. However, since in this case INT returns the next highest integer keep your wits about you when applying INT to negative values. For example:

```
5 REM **************
10 REM*** INT 2 ***
15 REM**************
20 LET Y=INT(-25.9)
30 PRINT Y
```

will return 26.

RELATED KEYWORDS: none

LABEL

Used within Assembly code as an address pointer. For example:

JP LABEL

means jump (or go to) the LABEL and carry out the operation stated after LABEL, such as:

LABEL:NOP

NOP meaning no operaton to be carried out.

LEFT$
SYNTAX : **v$=LEFT$(a$,i)**

ABBREVIATION : **LEF.**

This command facilitates the extraction of the left-most section of a string of characters.

Along with its related commands, RIGHT$ and MID$, LEFT$ enables a string to be sliced into smaller groups of characters. In the syntax example above, a$ is known as the 'target string', and LEFT$ will return a 'sub-string' of a$.

The target string can be a quoted group of characters or a string variable. Thus:

PRINT LEFT$("BATMAN",3)

and

PRINT LEFT$(A$,3)

are both legal statements.

The number of characters to be extracted is specified by the second parameter (i) and must be within the range 0-255. When executing a LEFT$ statement, the computer counts the characters within the string in question from left to right (with the left-most character in a$ counted as 1), until i characters have been dealt with. If i is greater than the number of characters in the string then the entire string will be returned. If i is 0, then a null string will be returned and if i is negative an 'Out of range' error will stop the program.

```
10 REM ***************
13 REM ***  LEFT$  ***
15 REM ***************
20 LET A$="MEMOTECH"
30 FOR I=1 TO LEN (A$)
40 PRINT LEFT$(A$,I)
50 NEXT I
```

RELATED KEYWORDS: RIGHT$, MID$, LEN

LEN
SYNTAX : **n=LEN (a$)**

ABBREVIATION : **none**

A function which returns the number of characters held in its argument (a$).

The LEN function is used when it is necessary to know how many ASCII characters are contained in a particular string. Usually LEN is applied to a string variable, although the function can also calculate the number of characters in a string which is literally quoted (i.e. PRINT LEN ("STRING")). The integer value returned by LEN will be in the range 0-255.

It is important to note that there must be a space between the LEN keyword and the first bracket, otherwise a 'Not numeric' error report will be generated. An important use of the LEN function is in the creation of

tabular screen displays:

```
10 REM ************
20 REM ***  LEN  ***
30 REM ************
40 CSR 10,3: PRINT "INPUT FIVE NUMBERS"
50 FOR A=1 TO 5
60 CSR 12,4+A: PRINT "NUMBER";A;
70 INPUT N$: LET N=VAL(N$)
80 CSR 20,4+A: PRINT CHR$(5)
90 LET L=LEN (N$)
100 IF N=0 THEN  GOTO 60
110 CSR 33-L,4+A: PRINT N$
120 NEXT A
```

NOTE: Line 80 is equivalent to positioning the cursor and pressing EOL.
Since LEN can only be applied to strings, numeric data can only be operated upon by this function if it is used in conjunction with STR$ (which converts numeric data into strings). If LEN is applied to a null string (a$=""), zero is returned.

RELATED KEYWORDS: MID$, LEFT$, RIGHT$, STR$

LET
SYNTAX : **LET v=n**
: **LET v$="string"**

ABBREVIATION : **LE.**

A statement which is used to assign values to variables.

LET statements are one of the most common constructions in BASIC. If we want to assign a value to a variable, for example:

V=2.8

this is achieved by means of the LET statement. So the following format is required:

10 LET V=2.8

The actual assignment of variables is fairly straightforward. For example:

20 LET W=19

simply instructs the MTX to hold the value 19 in a memory location which can be accessed via a variable called W. The value of W can be a redefined later in the program. For example:

50 LET W=W+10

tells the MTX to locate memory location W and add 10 to the value it already holds.

Assigning string variables works on the same principle as the definition of numeric variables, but the string variable name must be followed by a dollar sign ($) and the text that is being assigned to the variable must

be within quotes, thus:

```
60 LET A$="HELLO"
```

RELATED KEYWORDS: none

LIST

SYNTAX : **LIST**
: **LIST ln,ln**
: **LIST ln**

ABBREVIATION : **L.**

A command which displays the line(s) specified by ln of the BASIC program currently held in memory.

LIST is essential during the development of a program since it allows you to examine a single program line, a series of lines or the entire program.

The command can be entered in a variety of formats. For example:

```
LIST
```

on its own will display to the screen the entire program currently in memory. If the program fills more than a screenful of code the display will scroll upwards until it reaches the end of the listing. If you want to pause the scrolling at any point you can do so by pressing the PAGE key. This will sound a bell which will be repeated at regular intervals. To restart the scrolling press the PAGE key again, or if you want to stop the scrolling press the BRK key.

```
LIST 100,100
```

will display line 100 on the screen.

```
LIST 100,200
```

will display the lines 100-200 inclusive.

```
LIST 100
```

will list the lines from 100 to the end of the program.

RELATED KEYWORDS: EDIT

LLIST

SYNTAX : **LLIST i,i**
LLIST
LLISTi

ABBREVIATION : **LL.**

Lists the program currently in memory to the printer.
LLIST is normally entered as a direct command (there are rare occasions in which it is used in a program), and allows a program to be LLISTed to a printer. It works in exactly the same way as LIST. In other words:

```
LLIST
```

lists the entire program

```
LLIST 10
```

lists the entire program from line 10

10 LLIST 10,10

lists only line 10

10 LLIST 10,100

lists lines 10 to line 100

RELATED KEYWORDS: LIST,AUTO

LN
SYNTAX **: v=LN(n)**

ABBREVIATION **: none**

Function which returns base e logarithm of its argument.

LN (n) returns what is commonly referred to as the natural logarithm of n. The antilog of such a value can be calculated by using EXP(LN(n)). When necessary natural log operations can be used as with common logs. For example:

PRINT EXP(LN(N)+LN(V))

will print out the product of n and v

If n is a zero or a negative value the program will be halted with an 'Out of range' error message.

RELATED KEYWORDS: EXP

LOAD
SYNTAX **: LOAD ""**
: LOAD "filename"

ABBREVIATION **: LO.**

A command which LOADs a file in the MTX's memory from an external storage system.

Let's run through the syntaxes of this command to determine which format should be used when! When LOAD is used on its own as a direct command (LOAD "") the MTX will LOAD the first program it encounters on the cassette. However, if you use the LOAD "filename" syntax the computer will scan the cassette looking for the specified program and once it has been found it will be LOADed into memory.

For further information about this command refer to chapter 5.

RELATED KEYWORDS: SAVE, VERIFY

LPRINT
SYNTAX **: LPRINT a$**
: LPRINT i

ABBREVIATION **: LP.**

Allows the computer's PRINT facilities to be directed at a printer rather than the VDU.

The LPRINT statement uses exactly the same formats as the PRINT statement to precisely to same effect. The only difference between to two commands is that while PRINT produces a display on the screen, LPRINT determines and formats displays on a printer.

Creative use of LPRINT in conjunction with selected control codes (see Appendix X), allows the regulation of output to a variety of printers.

RELATED KEYWORDS: PRINT, LIST, LLIST, GR$

MID$
SYNTAX : MID$(A$,i,j)

ABBREVIATION : MI.

A string function which extracts a specified number of characters from a specified start point in a predefined string.

Like LEFT$ and RIGHT$, the string function MID$ is used to perform string slicing processes. In some respects the name of this keyword obscures its flexibility, since MID$ actually permits the extraction of characters from any part of a string, not just the middle.

The numeric parameters that follow the string variable in brackets must fall in the range 0-255. The first parameter (i) establishes the start point of the string to be extracted and the second (j) the number of characters to be removed.

In the syntax statement above, if i is greater than the length of the specified string a null string will be returned. If the second parameter (j) is greater than the length of the string the entire string (starting at i) will be returned.

Let's take a look at some possible formats for MID$. Say our source string is:

```
LET SOURCE$="MICROWAVE OVEN"
```

Then:

```
LET SUB$=MID$(SOURCE$,1,5)
PRINT SUB$
```

will PRINT out MICRO, and,

```
LET SUB2$=MID$(SOURCE$,11,4)
LET SUB3$=MID$(SOURCE$,3,1)
PRINT SUB3$+SUB2$
```

will PRINT out COVEN.

RELATED KEYWORDS: LEFT$, RIGHT$, LEN

MOD
SYNTAX : v=MOD(n1,n2)

ABBREVIATION : MO.

Numeric function which returns the remainder of the division of n1 by n2.

MOD allows us to establish the remainder of a division. It's implementation is quite straightforward. For

example, the statement:

```
10 LET R=MOD(3,2): PRINT R
```

will return 1 (since 3 divided 2 is 1 with a remainder of 1).

If the division of MOD's parameters produce a result which is less than 1, the function will return the full value of the number which is being divided. For example:

PRINT MOD(25,30)

will return 25.

The function will accept both positive and negative values. For example:

PRINT MOD(-3,2)

will return -1.

In the example program below we've used MOD to get around the problem of INT rounding to the next smallest integer. For example, if we require an integer value result from a division it's not inconceivable that we'll be looking for the nearest integer to the result (not the next smallest). For example:

PRINT INT(3.85/2)

will return 1. However, since 3.85 divided by 2 is 1.925, 2 is clearly the nearest integer value.

```
10 REM **************
20 REM ***  MOD  ***
25 REM **************
30 GOSUB 500
40 CSR 10,1: PRINT "WHEN CALCULATING X/Y:"
50 GOSUB 600
60 GOSUB 700
70 IF Y-M<Y/2 THEN  LET I(3)=I(2)+1 ELSE  LET I(3)
=I(2)
80 CSR 12,3: PRINT "IF X=";X$;" AND Y=";Y$
90 CSR 0,6
100 GOSUB 800
110 STOP
500 REM *** VARIABLES ***
510 DIM P$(3,10): DIM I(3)
520 LET S$=CHR$(11)+CHR$(5)
530 RETURN
600 REM *** INPUT ***
610 FOR A=1 TO 2
620 CSR 10,3: PRINT "ENTER A VALUE FOR ";
```

```
630 IF A<>1 THEN  PRINT "Y"; ELSE  PRINT "X";
640 INPUT X: PRINT S$
650 LET P$(A)=STR$(X)
660 IF SGN(X)<>-1 THEN  LET P$(A)=MID$(P$(A),2,LEN
 (P$(A)))
670 NEXT A
680 RETURN
700 REM *** VARIABLES ***
710 LET X$=P$(1): LET Y$=P$(2)
720 LET X=VAL(X$): LET Y=VAL(Y$)
730 LET M=MOD(X,Y): LET I(1)=X/Y
740 LET I(2)=INT(I(1)): LET H$="_"
750 LET R$=X$+"/"+Y$+"="
760 RETURN
800 REM *** RESULT ***
810 FOR B=1 TO 3
820 READ F$,G$
830 LET L=LEN (F$): LET R=LEN (R$)
840 GOSUB 900
850 PRINT
860 NEXT B
870 RETURN
900 REM *** PRINT ***
910 PRINT F$
920 FOR E=1 TO L: PRINT MID$(G$,E,1);: PAUSE 100:
NEXT E
930 FOR H=1 TO 19-L: PRINT H$;: PAUSE 80: NEXT H
940 FOR J=1 TO R: PRINT MID$(R$,J,1);: PAUSE 100:
NEXT J
950 PRINT I(B)
960 RETURN
1000 REM *** DATA ***
1010 DATA ORDINARY,DIVISION,USING,INT FUNCT
1020 DATA WITH MOD TO,CALL NEAREST INTEGER
```

RELATED KEYWORD: INT

MVSPR
SYNTAX : **MVSPR p,n,d**

ABBREVIATION : **MV.**

A flexible sprite command which alters certain attributes of a previously activated sprite.

The command's first parameter p determines the function to be performed and the number n is the sprite number. The range of values for d depends upon the value of p.

p	FUNCTION	d
1	Move 1 step in direction d	0-8
2	Select sprite pattern d	0-127 (8*8 sprites) 0-31 (16*16 sprites)
4	Change to direction d	0-8
8	Plot a point at centre of sprite	-

Any one of the first three functions (p=1,2 or 4) can be used in conjunction with the fourth function, p=8.

For further information on this command refer to the entry for ADJSPR and chapter 10.

RELATED KEYWORDS: ADJSPR, CTLSPR, GENPAT, SPRITE

NEW
SYNTAX : **NEW**

ABBREVIATION : **none**

A command that resets BASTOP, NBTOP, ARRTOP, BASTP0 and calls CLEAR.

As should be obvious from the definition above, NEW is probably the most uncompromising of the BASIC keywords, since it completely wipes any BASIC program from memory. It also clears all the program's variables.

It is advisable to enter NEW as a direct command before starting any new program to ensure that it is unaffected by any program lines left over from a previous listing.

RELATED KEYWORDS: RUN

NEXT
SYNTAX : **NEXT v**

ABBREVIATION : **N.**

Statement which determines the end of a FOR...NEXT loop.

MTX BASIC allows the loop variable to be omitted from a NEXT statement, although its inclusion does clarify a listing (especially in the case of a nested loop structure).

For a description of FOR...NEXT statements see the entry under FOR.

RELATED KEYWORDS: FOR, TO, STEP

NODDY
SYNTAX : **NODDY**
ABBREVIATION : **NODD.**

The command used to access the Noddy language.

This command's sole function is to access the Noddy language and therefore it is usually entered as a direct command. It can be used within a program but will stop any further BASIC processing.

It is useful to use the NODDY command in conjunction with PLOD when you are developing a Noddy program, since all Noddy programs *RETURN to BASIC once they have been executed. However, by using the following program you can return directly to Noddy.

```
10 REM ***************
13 REM ***  NODDY  ***
15 REM ***************
20 PLOD "FILE"
30 NODDY
```

This program will PLOD the program called FILE (PLOD is Noddy's equivalent to the BASIC language's RUN statement) and once it has been processed it will return control to the Noddy language.

For further information on the NODDY command and the Noddy language refer to chapter 11.

RELATED KEYWORDS: PLOD

NOT
SYNTAX **: NOT (c)**

ABBREVIATION **: none**

Logical operator which inverts the logical value of the condition to which it is applied.

Since the MTX's logical operators do not facilitate bit-wise logical operations, and thus NOT can only be applied to conditional expressions, its application is very straightforward. NOT simply reverses the logical value of any condition it preceeds. For example:

```
10 REM ***************
13 REM ***   NOT   ***
15 REM ***************
20 LET A=6: LET B=6
30 LET N=( NOT A=B)
40 PRINT N
```

will return 0, since the condition A=B is True, and thus the value created by NOT True is False (0). Now change line 30 to:

```
30 LET N=( NOT A>B)
```

and the program will return -1. since the expression A>B is False, which inverted by NOT assigns the computer's True value to N (-1). It's easy to see how this operator could be used to determine whether or not a program is to be re-RUN. For example:

```
10 REM ***************
13 REM ***  NOT 2  ***
15 REM ***************
200 INPUT "AGAIN?";A$
210 IF  NOT A$="Y" THEN  STOP
220 PRINT "REPEAT PROGRAM"
230 GOTO 200
```

So unless A$="Y" is True, the computer will execute what is ironically the IF...THEN statement's True option (i.e that A$ is NOT "Y") and STOP the program.

RELATED KEYWORDS:AND,OR,IF,THEN,ELSE

ON

SYNTAX **: ON v GOSUB ln, ln,...**
: ON v GOTO ln, ln,...

ABBREVIATION **: O.**

Statement which when used in conjunction with GOSUB or GOTO, allows the flow of control to be determined by the value of v and directed to one of the number of specified line numbers.

This statement often makes an appearance in menu-driven programs in which the user's input determines which of a number of processes are executed by the computer. For example:

```
10 REM ************
13 REM ***  ON  ***
15 REM ************
20 INPUT "ENTER 0, 1 OR 2 ";V
30 ON V GOSUB 60,70,80
40 STOP
50 REM ********************
53 REM ***  SUBROUTINES  ***
55 REM ********************
60 PRINT "ZERO": RETURN
70 PRINT "ONE": RETURN
80 PRINT "TWO": RETURN
```

The use of the ON statement in line 30 will call the subroutine in line 60 if 0 is entered, line 70 if 1 is entered and if 2 is keyed in the line 80 is executed. If V is not an integer then the INPUT will be rounded to the next lowest integer. If the INPUT is negative an 'Out of range' error is generated. The rest of the ON statement will be ignored if the value of V exceeds the number of specified line destinations. For further examples see the entry for GOSUB.

RELATED KEYWORDS: GOTO, GOSUB, RETURN

OR
SYNTAX : **c OR c**
: **e OR e**

ABBREVIATION : **none**

OR is a logical operator which performs logical disjunction on conditions.

For conditional expressions linked by OR, the combined expression is evaluated as either True or False as in the truth table below:

Condition1	Conditional operator	Condition2		C1 OR C2
TRUE	OR	TRUE	=	TRUE
FALSE	OR	TRUE	=	TRUE
TRUE	OR	FALSE	=	TRUE
FALSE	OR	FALSE	=	FALSE

Thus the result of two expressions combined using OR is True if either of both conditions are True, and False only if both conditions are False. OR is useful in checking whether values are within acceptable ranges. For instance, take the case of an input for the month:

```
200 INPUT "MONTH (1-12) ";M
210 IF M<1 OR M>12 THEN  PRINT "WHAT?!!! TRY AGAIN ": GOTO 200
```

RELATED KEYWORDS: AND, NOT

OUT
SYNTAX : **OUT p,v**

ABBREVIATION : **OU.**

Input/Output statement which sends a specified value (v) to the port whose code is established by the statement's first parameter (p).

RELATED KEYWORDS: LPRINT, LLIST, BAUD

PANEL
SYNTAX : **PANEL**

ABBREVIATION : **PAN.**

A command that gives you a 'window' on the memory and registers.

By simply entering PANEL as a direct command eight registers are listed along with the addresses that they hold and the contents of those addresses.

These appear at the top right of the screen, at the bottom a block of memory with its hexadecimal contents is listed. By entering I2000 a block of assembly code is listed: this is the code the computer has used to create the PANEL.

The PANEL can be utilised in a multitude of ways but its chief advantage is that it allows you to test programs by showing the current state of the registers one stage (instruction) at a time.

RELATED KEYWORDS: PEEK

PAPER
SYNTAX : **PAPER p**
BASIC Token :
ABBREVIATION : **PA.**

Sets a background colour for the MTX's text screen, which is determined by the statement's argument (p).

PAPER utilises exactly the same colour codes as the INK statement, and you should refer to the INK entry for a full list of the available pigments. In text mode it sets the background colour for the entire screen, and cannot be used to colour individual sections of screen. The example program below demonstrates all the possible PAPER variations in conjunction with the various INK combinations.

For further discussion of both PAPER and INK, see the graphics section (chapter 8).

```
5 REM  **************
10 REM ***  PAPER  ***
15 REM **************
20 FOR P=1 TO 15
30 PAPER P
40 CSR 1,12: PRINT "PAPER ";P
50 FOR I=1 TO 15
60 INK I
70 CSR 1,4: PRINT "INK ";I
80 CSR 16,5+I: PRINT "INK"
90 PAUSE 300
100 CSR 16,3+I: PRINT CHR$(5)
110 NEXT I
120 PAUSE 1000: CLS
130 NEXT P
```

RELATED KEYWORDS: INK, COLOUR

PAUSE
SYNTAX : **PAUSE n**

ABBREVIATION : **PAU.**

Halts the execution of a program for a specified period of time.

The length of time that a program is PAUSEd is dependent upon the value assigned to the PAUSE statement. This value must be within the range of 0-65535, and the larger the number the longer the wait. It is not possible to achieve precise time delays with the statement since there will always be slight variation determined by the processes being executed at the time. This said it is fairly safe to assume that a PAUSE of one second can be achieved by assigning n the value of 1000.

```
10 REM **************
15 REM ***  PAUSE  ***
```

```
17 REM ***************
20 FOR A=1 TO 10
30 PRINT A
40 PAUSE 2500
50 CLS
60 PAUSE 1000
70 NEXT A
```

RELATED KEYWORDS: none

PEEK
SYNTAX : PEEK addr.

ABBREVIATION : PE.

Statement which examines the contents of a specified memory location (addr.) and returns the value contained in that location.

The value returned by PEEKing one of the MTX's memory locations will fall in the range 0-255, while the address specified by the statement must be in the range 0-65535 if an 'Out of range' error is to be avoided. The MTX's PANEL facility offers users an unusually sophisticated tool for the examination of memory, but PEEK is a more conventional BASIC statement which offers a more convenient method of examining a single location.

RELATED KEYWORDS: POKE,PANEL

PHI
SYNTAX : PHI x

ABBREVIATION : PH.

This function is used to alter the plotting direction.

The value of PHI's parameter (x) is the angle of change, measured in radians (see ANGLE). Positive values of x change the plotting direction anti-clockwise, negative values cause clockwise rotation. The following example DRAWs a ten sided figure on the screen:

```
1 REM **************
3 REM ***  PHI  ***
5 REM **************
10 VS 4: CLS
20 ANGLE 0
30 PLOT 127,95
40 FOR T=1 TO 10
50 DRAW 25
60 PHI PI/5
70 NEXT T
80 GOTO 80
```

For further information on this command refer to chapter 9.

RELATED KEYWORDS:

PI
SYNTAX **: PI**

ABBREVIATION **: none**

The trigonometric constant which returns a value of 3.14159265.
The computer stores an accurate value of PI so that it doesn't have to be recalculated each time that it is needed. In the following example program, PI is used in the calculation of the area of a circle:

```
1 REM ************
3 REM ***  PI   ***
5 REM ************
10 VS 5: CLS
20 FOR V=1 TO 5
30 LET R=INT(RND*30)
40 LET A=PI*(R^2)
50 PRINT "IF CIRCLE RADIUS =";R
60 PRINT "THEN ITS AREA IS";A
70 PRINT
80 PRINT
90 NEXT V
```

RELATED KEYWORDS: ANGLE, PHI

PLOD
SYNTAX **: PLOD "filename"**

ABBREVIATION **: PL.**

The command used to run a Noddy program.

In the same way that RUN is used to execute a BASIC program, the PLOD command is used to execute a Noddy program. Before you can PLOD a Noddy program you must be in BASIC and then you can use the command either within a program or as a direct command. In either case the command must be followed by the name of the program enclosed within quotes.

For further information about the PLOD command refer to chapter 11.

RELATED KEYWORDS: NODDY

PLOT
SYNTAX **: PLOT x,y**

Plots a point on the graphics screen at the co-ordinate x,y.

The graphic screen measures 256 (0-255) in the horizontal (x) direction by 192 (0-191) vertically (y). The

co-ordinate 0,0 is at the bottom left corner of the screen. The following program PLOTs a dotted line across the centre of the screen:

```
1 REM ************
3 REM ***  PLOT ***
5 REM ************
10 VS 4: CLS
20 FOR T=0 TO 255 STEP 4
30 PLOT T,96
40 NEXT T
50 GOTO 50
```

For further information on this command refer to chapter 9.

RELATED KEYWORDS: DRAW, ANGLE, PHI, GR$

POKE
SYNTAX : **POKE addr., i**

ABBREVIATION : **PO.**

Statement which enables a single byte binary value (i) to be placed into the memory location specified by the statement's first parameter (addr.).

For the majority of programming demands, the BASIC language is a perfectly adequate means of solving problems with your MTX. However, there are circumstances which require a greater level of control than BASIC can provide, but do not justify the development of customised assembly language routine. This is where POKE statements come into their own.

The first parameter (addr.) which specifies the location must be in the range 0-65535 and the value to be placed into the location (i) must be an integer in the range 0-255. If the first parameter falls outside this range the program will continue, but no value will be stored anywhere in memory! If the second parameter (i) is greater than 255, the computer will subtract 255 from the value specified and, starting from zero, will store the remainder. Thus POKE 50000,256 will store 0, and POKE 50000,356 will store 100.

RELATED KEYWORDS: PEEK, PANEL

For further information on PANEL see chapter 12.

PRINT
SYNTAX : **PRINT plist plist = list of print items**

ABBREVIATION : **P.**

Statement to display PRINT items to the screen.

The items to be displayed can be literally quoted strings which must be centred within double quotes:

```
PRINT "A STRING"
```

string variables:

```
LET AS="STRING VARIABLE" : PRINT AS
```

literally quoted numeric data, which do not require quotes:

PRINT 2*45

and numeric variables:

LET A=32/7 : PRINT A

PRINT used by itself without any PRINT items displays a blank line on the screen. The PRINT statement is described at some length in chapter 2 and further clarified in chapter 8, so we shall merely outline the possible formats here:

i) PRINT items to be displayed literally to the screen must be enclosed with quotes, (e.g. PRINT "HJ&^ 54")

ii) Any arithmetical expression to be calculated must not be included within quotes (e.g. PRINT 2.56*4)

iii) Semi-colons after PRINT items leave the PRINT position set directly after the last character PRINTed, so that the next item to be PRINTed will follow on. For example:

```
10 PRINT "A STRING AND ";
20 PRINT "ANOTHER STRING"
```

will display A STRING AND ANOTHER STRING.

iv) The MTX's screen is divided into five PRINT fields. Including a comma between the PRINT items causes the item following a command to start PRINTing at the first position of the next PRINT field:

10 PRINT 1,2,3,4

v) PRINT can also be combined with the CSR command. For example,

CSR 10,10: PRINT "FRED"

will PRINT the literally quoted string "FRED" on the screen at the co- ordinates 10,10.

RELATED KEYWORDS: CSR

RAND
SYNTAX : RAND

ABBREVIATION : RA.

Numeric function which sets the seed value for the MTX's RaNDom number generator.

RAND is used in conjunction with RND in the creation of random numbers. If you RESET the MTX and then RUN the following routine you'll be confronted with a sequence of five apparently random numbers in the range 0-0.9999999.

```
5 REM  **************
10 REM ***  RAND  ***
15 REM **************
20 FOR A=1 TO 5
30 PRINT RND
40 NEXT A
```

RESET the machine again and enter and re-RUN the program. You'll discover that the RND function isn't so random after all! In fact the computer will produce exactly the same numerical sequence. This is because RND uses the computer's random number generator to produce values, and the generator creates a numerical sequence by performing calculations on a base number. Unless it is otherwise instructed, it always uses the same base for its calculations (the basis for these calculations is known as the random number generator's "seed value"), and consequently always produces the same sequence of numbers.

RAND is the function which enables us to specify RND's seed value. This facility enables the creation of predictable sequences of "random" numbers whose values are determined by the specified seed value, but which are always the same whenever that value is used. To test this out, RUN our example program once again, this time adding the following line:

```
15 RAND 200
```

You'll discover that the computer generates a completely different sequence of "pseudo-random" values, which will be repeated each time the program is RUN. This is because line 15 uses the RAND statement to re-establish the specified seed value each time the program is RUN.

In order to create genuinely random sequences of random numbers the RAND value must be negative. Try altering line 15 to:

```
15 RAND -10
```

and RUN the program repeatedly.

For further information about controlling the range of values produced by the random number generator, see the entry for RND.

RELATED KEYWORDS: RND

READ
SYNTAX **: READ v$,v$,v$...**
: READ v,v,v...
: READ v$,v,v$,v...

ABBREVIATION **: REA.**

This statement is used to load the next DATA item into the variable(s) specified by READ.

The variable names which follow a READ statement can be string and numeric. However, the DATA which is to be loaded into the READ variable(s) must be appropriate to the variable in question (i.e. string DATA assigned to a string variable and numeric DATA to a numeric variable) or a 'Mismatch' error will be generated.

When the DATA pointer in the MTX's memory has reached the last DATA item, the computer will generate a 'No data' error if any further attempt is made to READ DATA without prior use of the RESTORE statement. RESTORE will return the data pointer to a specified DATA statement. For further details about the use of this command, see the entry for DATA.

RELATED KEYWORDS: DATA, RESTORE

REM
SYNTAX **: REM statement**

ABBREVIATION : R.

A statement which allows REMarks to be inserted into a program without a 'Mistake' error being generated or the program being in any way affected.

These statements are of immense value to all programmers. Whilst their inclusion has no effect on a listing when it is actually RUN, the function of the statement is to provide an internal commentary which explains the action of a particular line or section of code. REM statements are normally included at the beginning of a line, for example:

150 REM * SUBROUTINE 1 ***

they can also appear at the end of a multi statement line:

180 LET A=PI*R ^ 2: REM * AREA OF A CIRCLE ***

The important point to remember is that the computer will ignore all the instructions following a REM statement for the duration of the line in which it is included.

There is a tendancy to only use REM statements when a program becomes especially complex or inpenetrable, on the grounds that short code will be self-explanatory. However, it can be argued that REMs should always be used in programs, however brief or straightforward, since even a schematic commentary of REMs can avoid many wasted hours of searching through unmarked listings.

RELATED KEYWORDS: none

RESTORE
SYNTAX : RESTORE ln

ABBREVIATION : RES.

Statement which allows the contents of DATA statements to be READ more than once.

When processing BASIC programs, the MTX uses an internal pointer to keep track of the last DATA item READ into the DATA variable(s). Under normal circumstances, once a DATA item has been READ it cannot be READ again. However, the RESTORE statement resets the internal pointer to the specified line, thus enabling the DATA to be used from that point onwards. For more details about this statement see the entries for DATA and READ.

RELATED KEYWORDS: DATA, READ

RETURN
SYNTAX : RETURN

ABBREVIATION : RET.

Statement used to mark the end of a subroutine and RETURN control to the main program.

When a subroutine is called from the main body of a program, control is passed to the line number specified by the GOSUB statement. When the instructions contained in the subroutine have been processed a RETURN statement is required to return control to the line that follows the original GOSUB statement. A RETURN without a GOSUB statement will halt the program with an error report. Nested subroutines require a RETURN for each GOSUB statement.

For further details about subroutines see the entries for GOSUB and ON.

RELATED KEYWORDS: GOSUB, ON

RIGHT$
SYNTAX : **v$=RIGHT$ (a$,i)**

ABBREVIATION : **RIG.**

A string function which enables a sub-string of a specific length to be extracted from the right-most end of a predefined string.

Like MID$ and LEFT$, RIGHT$ facilitates the creation of a sub-string from a predefined string. It takes the form:

```
LET SUB$=RIGHT$(a$,i)
```

in which SUB$ is the variable name to which the new string will be assigned, and a$ is the source string from which i characters will be extracted from the right-most end. i can be any value from 0-255 and a$ either a predefined variable or a literally quoted string within the brackets, (RIGHT$("RICHARD",4)) would return HARD. If i is 0 a null string will be returned, and if i is greater than the length of the source string (a$) then the entire string will be returned. The following example should clarify the operation of this function:

```
10 REM ****************
13 REM ***  RIGHT$  ***
15 REM ****************
20 LET A$="RIGHTMOST"
30 LET SUB$=RIGHT$(A$,4)
40 PRINT SUB$
```

will PRINT out MOST to the screen. For more details about string slicing see the entries for LEFT$ and MID$, LEN.

RELATED KEYWORDS: LEFT$, MID$, STR$

RND
SYNTAX : **v=RND**

ABBREVIATION : **RN.**

Numeric function that generates a floating point value between 0 and 1.0.

Used in conjunction with RAND, RND is one of the MTX's most valuable numeric functions. It allows us to create code that produces (ostensibly) random numeric data, which in turn means that we can introduce an unpredictable element into our programs. (Think how tedious a game would be if the Invader/Spaceship/Ghost always emerged from the same screen position !)

The values produced by RND are calculated by the MTX's random number generator. As we shall discover, these numbers are actually only "pseudo-random", but generally this qualification does little to detract from the statement's value. The MTX creates its RaNDom numbers by performing a series of calculations on a base number known as the random number generator's "seed value". On power-up this seed value is always the same, and thus the sequence of values which can be accessed by RND are always the same. To test this out, switch your machine off and then on again and enter/RUN the following program:

```
10 REM ****************
13 REM ***    RND     ***
```

```
15 REM ****************
20 FOR A=1 TO 5
30 PRINT RND
40 NEXT A
```

The following sequence will be displayed on the screen:

6.25610352E-04
0.919647217
0.962387085
0.747253418
0.715896606

At first glance RND appears to be doing its job. The values produced don't appear to have anything in common, and could thus be described as random. If you RUN the program again, this initial impression appears to be confirmed. The computer will display yet another sequence of "random" values. But appearances can be deceptive! Switch your micro off and on again. Re-enter and RUN the example. You'll be confronted with exactly the same (pseudo) random sequence!

We can alter the sequence of numbers generated by the MTX by using the RAND function. Add the following line to the example above:

```
15 RAND 1000
```

The RAND statement sets a seed value of 1000 on which the random number generator will perform its calculations, and thus a different sequence of values will be display. Each positive value in a RAND statement produces a different RND sequence, but each time that value is used the same sequence will be produced. However, if RAND's argument is negative, the numerical sequence will appear genuinely random. Try changing line 15 to:

```
15 RAND -1000
```

and repeatedly RUN the program. You'll get a different values on each occasion. So RND's "pseudo" randomness can be improved upon with very little effort. But is a random value between 0 and 1 of much use to us ? Well, not very often, but there are ways in which the range of values produced by RND can be extended. Let's say we want to produce a dozen random integers in the range 0-255.

```
10 REM ****************
13 REM ***   RND 2   ***
15 REM ****************
20 RAND -100
30 FOR A=1 TO 12
40 PRINT INT(RND*256)
50 NEXT A
```

Note that the integer value produced by line 40 multiplies RND by 256, even though our specified range 0-255. This is because although the value produced by RND can equal 0 it never quite reaches 1.

Let's look another routine which demonstrates how we can set both an upper and lower limit for our random sequence:

```
10 REM ****************
13 REM ***   RND 3   ***
15 REM ****************
20 RAND -30
30 FOR A=1 TO 12
40 PRINT INT(RND*81)+175
50 NEXT A
```

This program prints out twelve random integer values in the range 175- 255. The way in which this program sets the numerical limits should be obvious. Take a look at line 40. The difference between our upper and lower limit is 80 (255-175). Since we have decided that our lower limit is 175, and 0 occasionally turns in a RND sequence (and thus RND*0=0), we have to add 175 to whatever value is produced to ensure that our result never falls below the lower limit. Since we have discovered that 0.9999999 is the largest value that we can obtain from RND and we're after an integer sequence, it's no good multiplying RND by 80, since this calculation would never produce our lower range value (the largest number we could hope for would be:

80*0.9999999 = 79.999992

which the INT function would round down to 79). This is why we have used RND*81, which assuming a maximum random value can produce 80.9999919, which INT will round down to 80.

RELATED KEYWORDS: RAND.

RUN
SYNTAX : **RUN**

ABBREVIATION : **RU.**

A command which clears all variables and commences execution of the BASIC program currently in memory.

RUN is used as a direct command to start the execution of the BASIC program you have just LOADed or keyed in. It can also be used within a program. For example:

```
1 REM ************
3 REM ***  RUN  ***
5 REM ************
10 INPUT "DO YOU WANT ANOTHER GO? (Y/N)?";C$
20 IF C$="Y" THEN  RUN
```

RELATED KEYWORDS: CLEAR, CONT, NEW

SAVE
SYNTAX : **SAVE ""**
: **SAVE "filename"**
BASIC Token :

ABBREVIATION : **SA.**

A command that stores a file onto tape.

This command is used to SAVE a program or file on to a cassette. The first syntax above should never be used, since you should always give the program yhat you wish to SAVE a filename by which it can be traced. The filename can be of any length, although obviously should be kept as short as possible to make it easy to enter and remember.

RELATED KEYWORDS: LOAD, VERIFY

SBUF
SYNTAX : **SBUF n**

ABBREVIATION : **SB.**

Reserves space in memory to create sound buffers.

This command is similar to DIM in that it is used to reserve space to create sound buffers in the memory for use by the continuous sound command structure. The MTX automatically reserves space for one sound buffer as you only need declare Sound BUFfers if you intend to use more than one continuous SOUND command.

The maximum number of declarable buffers is 255, where each buffer declared reserves 12 bytes of memory per channel. Since 4 channels are available, the statement:

10 SBUF 20

will reserve 960 bytes (4 channels * 12 bytes * 20 Sound BUFfers or 4*12*20=960). By allowing the sound chip to retrieve the latest values contained in the buffers, the control processor is freed to perform other tasks.

NB. If sound buffers are used in conjunction with the single disc system, the should be limited to 7 buffers per channel. If more than 7 are used, you will need to reset the computer before any subsequent disc accesses can be performed.

RELATED KEYWORDS: SOUND

SGN
SYNTAX :**v=SGN(n)**

ABBREVIATION : **SG.**

A numeric function which returns a value which indicates the sign or signum of its argument.

The SGN function returns 1, 0 or -1. It is used to indicated whether its argument is positive, negative or zero. If n is positive then v=1, if n is negative then v=-1 and if n=0 then v=0. The following example program demonstrates its use in a rather contrived routine:

```
10 REM ************
13 REM ***  SGN  ***
15 REM ************
20 CLS : PRINT "ENTER A NUMBER"
30 INPUT N
40 PRINT "YOUR NUMBER WAS ";N
50 ON SGN(N)+1 GOSUB 100,200,300
60 INPUT "ANOTHER GO (Y/N)? ";D$
70 IF D$="Y" THEN  GOTO 10
80 STOP
```

```
100 PRINT "A NEGATIVE VALUE": RETURN
200 PRINT "ZERO": RETURN
300 PRINT "A POSITIVE VALUE": RETURN
```

By adding 1 to the value returned by the SGN statement in line 50, control is directed to the subroutine which prints the appropriate status of the value of the INPUT (i.e. -1+1 = 0 = GOSUB 100; 0+1 = 1 = GOSUB 200; 1+1 = 2 = GOSUB 300).

RELATED KEYWORDS: ABS

SIN
SYNTAX : **v=SIN(n)**

ABBREVIATION : **SI.**

A numeric function which returns the sine of its argument (n)

Like all the MTX's trigonometric functions, SIN returns a value measured in radians. To convert radians into degrees use the following formula, in which r equals the number of radians:

degrees = r * 180/PI

The following program returns the SINe of angles from 0 to 360 in steps of five degrees:

```
10 REM ************
13 REM ***  SIN  ***
15 REM ************
20 CLS
30 FOR A=0 TO 360 STEP 5
40 LET RAD=A*PI/180
50 PRINT A,SIN(RAD)
60 NEXT A
```

Notice the use of the degrees to radians conversion formula in line 40.

RELATED KEYWORDS: ATN, COS, TAN

SOUND
SYNTAX : **SOUND c,f,v,fc,vc,d,m**
: **SOUND c,f,v**

ABBREVIATION : **SO.**

The command used to create SOUNDs on the MTX.

This command can be used in two distinct ways. The type of application is determined by the number of parameters used. If the SOUND command only takes 3 parameters the MTX takes it to be a direct sound command. 7 parameters signify that the SOUND is to be extended through Sound BUFfers, and such a statement is called continuous SOUND.

The first three of the statement's parameters are common to both versions, where c=channel, f="frequency" and v=volume. However, this said, the "frequency" value does differ depending upon the type of command used (see chapter 7).

The additional 4 parameters are only used by the continuous SOUND command. These are fc="frequency" change, vc=volume change d=duration and m=mode. These parameters allow you to vary the volume and frequency, specify the amount of time the sound is to be enveloped and decide if the sound it to be linked to the preceding command.

Refer to chapter 7 for further information on the SOUND command.

RELATED KEYWORDS: SBUF

SPK$
SYNTAX : A=SPK$

ABBREVIATION : SPK.

SPK$ is the Screen-Peek function which peeks the character at the current cursor position and then auto-increments the cursor location.

The character returned by the SPK$ function is an ASCII code. The example program below PRINTs text onto the screen and then uses SPK$ to read the screen and store the text into the string variable A$.

```
1 REM **************
5 REM ***  SPK$  ***
7 REM **************
10 VS 5: CLS : DIM A$(30)
20 CSR 2,6: PRINT "TOO MANY COOKS SPOIL THE BROTH"
30 CSR 2,6
40 FOR T=1 TO 30
50 LET A$=A$+SPK$
60 NEXT T
70 CSR 2,12: PRINT A$
```

For further information on this command refer to chapter 10.

RELATED KEYWORDS: GR$

SPRITE
SYNTAX : SPRITE n,pat,xp,yp,xs,ys,col

ABBREVIATION : S.

The SPRITE command is used to define the characteristics of a particular sprite. See also CTLSPR.

SPRITE is used to set up the initial attributes for a sprite shape. The parameters used by the SPRITE command are:-

parameter	meaning	
n	sprite number	1 to 32

pat	pattern number	0 to 127 (8*8 sprite) 0 to 31 (16*16 sprite)
xp	x position	-4095 to +4095
yp	y position	-4095 to +4095
xs	x speed	-128 to +127
ys	y speed	-128 to +127
col	sprite colour	0 to 15

The example program places a black sprite in the centre of the screen.

```
1 REM ****************
3 REM ***  SPRITE  ***
5 REM ****************
10 VS 4: CLS
20 CTLSPR 2,1
30 GENPAT 3,0,60,126,219,255,231,126,36,60
40 SPRITE 1,0,127,96,0,0,1
50 GOTO 50
```

For further information on this command refer to chapter 10.

RELATED KEYWORDS:CTLSPR

SQR
SYNTAX : **v=SQR(n)**

ABBREVIATION : **SQ.**

Function which returns the square root of its argument.

Another of MTX BASIC's valuable numeric functions, SQR returns the square root of any positive number. However, if the value of its argument (n) happens to be negative, processing will be halted with an 'Out of range' error.

The simple example below shows the function in action:

```
10 REM **************
13 REM ***  SQR  ***
15 REM **************
20 CLS
30 FOR I=45 TO 60
40 PRINT "SQR ROOT OF";I;" IS";SQR(I)
50 NEXT I
```

RELATED KEYWORDS : ABS

STEP (SEE FOR)

STOP
SYNTAX : STOP

ABBREVIATION : ST.

A command that halts a program.

This command tells the computer to STOP a program. You can continue processing from the line following the STOP command by using CONT or GOTO ln.

```
5 REM  **************
10 REM ***  STOP  ***
15 REM **************
20 PRINT "WHEN THIS PROGRAM STOPS TYPE CONT"
30 PRINT "AND IT WILL CONTINUE TO RUN FROM "
40 PRINT "LINE 70"
50 PAUSE 5000: CLS
60 STOP
70 PRINT "THIS WILL BE PRINTED WHEN THE "
80 PRINT "PROGRAM IS CONTINUED"
```

RELATED KEYWORDS: CONT, GOTO, RUN

STR$
SYNTAX : a$=STR$(v)

ABBREVIATION : STR.

A string function which returns a string representation of its numeric argument (v)
STR$ is the function which enables string functions to be applied to numeric values by converting them into a string format. For example, when the MTX PRINTs a positive value to the screen it includes a leading space in the display. This can result in untidy screen displays when the following type of statement is used:

```
40 LET V=61
50 PRINT "VALUE=";V;" UNITS"
```

However, by using the STR$ function the display can be tidied up by using one of the string slicing functions (MID$ in this case):

```
20 LET V=61: LET A$=STR$(V)
30 LET B$=MID$(A$,2,2)
40 PRINT "VALUE=";B$;" UNITS"
```

The inclusion of the leading space should be borne in mind when the STR$ function is used in the comparison of strings. Thus, if '6510' and STR$(6510) were compared, the computer would include the leading space in the result of the latter and find the two strings unequal. STR$ performs the opposite function to VAL, which converts strings of numbers into their numeric value.

RELATED KEYWORDS: VAL

TAN
SYNTAX : **v=TAN(n)**

ABBREVIATION : **TA.**

A trigonometric function that returns the tangent of its argument (n).

Like all of the MTX's trionometric function. TAN calculates angles in radians (not degrees). Thus both its argument and the value returned when the TAN function is used are expressed in radians. To convert degrees to radians and vice versa, the following formulae must be used:

Degrees = r/PI*180 Radians = P/180

RELATED KEYWORDS: ATN, COS, SIN

THEN (SEE IF)

TIME$
SYNTAX : **TIME$**

ABBREVIATION : **TI.**

Stores the current value of the MTX's real time clock in hours, minutes and seconds.

TIME$ enables us to access the MTX's real time clock. Since it is the string form of a system variable it is not possible to directly assign a value to TIME$. If you use it without a CLOCK statement, TIME$ will return the time that has elapsed since power-up. So if you want to know how many hours you've been hacking away simply enter:

```
PRINT TIME$
```

The ability to discover how long your computer has been in action for may be interesting on occasion, but doesn't suggest that TIME$ is of enduring value. Thankfully, MTX BASIC also offers the CLOCK facility, which enables us to set a start value for the characters stored by TIME$. In other words, a TIME$ statement allows us to turn your micro into a (very expensive!) digital clock.

Both TIME$ and CLOCK use hours, minutes and seconds as their temporal parameters. Thus if we ask the computer to print TIME$ when it is storing 12:30 it will return:

```
123000
```

CLOCK works on the same principle. If we want to set the computer's internal clock to 12:30 we must enter:

```
10 CLOCK "123000"
```

Note that the micro's clock will run for a hundred hours before returning to "000000". Although TIME$ cannot be directly defined, as a string variable it can be subjected to string manipulation. Our example demonstrates how TIME$ can be sliced and recycled!

```
5 REM  ********************
10 REM
15 REM ***     TIME$     ***
20 REM
25 REM ********************
30 DIM B$(10): PAPER 1: CLOCK "000000"
40 GOSUB 200
50 FOR A=1 TO 5 STEP 2
```

```
60 CSR 9,7+A
70 PRINT MID$(TIME$,A,2)
80 NEXT A
90 GOTO 50
200 REM ******************
205 REM
210 REM *** SUBROUTINE ***
215 REM
220 REM ******************
230 PRINT "THIS PROGRAM HAS BEEN";
240 PRINT " RUNNING FOR: "
250 FOR B=8 TO 12 STEP 2
260 READ B$: CSR 12,B: PRINT B$
270 NEXT B
280 RETURN
300 DATA HOURS,MINUTES,SECONDS
```

RELATED KEYWORDS : CLOCK,PAUSE

TO (SEE FOR)

USR
SYNTAX : USR(addr)

ABBREVIATION : none

Function which calls a machine-code subroutine whose start address is established by the statement's argument (addr.).

The USR command is the traditional way by which machine-code subroutines are accessed from a BASIC program. Machine code programming is beyond the scope of this book, but the principle of a USR statement is quite strightforward. USR works in much the same way as GOSUB. It calls the machine-code routine starting at the location specified (addr.), and executes the code until it is instructed to return to BASIC.

RELATED KEYWORDS: CODE, PEEK, POKE

VAL
SYNTAX : v=VAL(a$)

ABBREVIATION : VA.

String function which returns the numeric value of the string of numbers contained in its argument
VAL performs the opposite function to STR$ (which converts a numeric value into its string representation). If the computer encounters a non-numeric character in the middle of the string, it will ignore the string from that point onwards. Thus:

```
10 REM ************
13 REM ***  VAL  ***
15 REM ************
20 LET N$="65XY10"
30 LET V=VAL(N$)
```

```
40 PRINT V
```

will PRINT 65 to the screen

```
10 REM ************
13 REM *** VAL 2 ***
15 REM ************
20 CLS
30 INPUT "ANY NUMBER ";V$
40 LET B$="5"
50 PRINT V$+B$: PRINT
60 LET V=VAL(V$): LET B=VAL(B$)
70 PRINT V+B
80 STOP
```

In the above example, line 50 performs a simple string concatenation (and thus PRINTs the input string next to the 5 contained in B$). However, after VAL has determined the numeric value of these strings (line 60), they can be added together and the result PRINTed out (line 70).

RELATED KEYWORDS: STR$

VERIFY

SYNTAX : **VERIFY "filename"**
: **VERIFY ""**

ABBREVIATION : **VE.**

An Input/Output command which makes it possible to ensure that a program has been correctly transferred to a tape.

There are few more frustrating surprises than the discovery that a program you have SAVEd has been incorrectly transferred to a tape. The VERIFY command offers programmers a means of ensuring that the program that has been transferred to an external storage device is exactly the same as the one held in the MTX's memory.

When the command is followed by a filename (which must be contained within quotes and be exactly the same filename as the one used in the SAVE statement), the computer will search the tape until it locates the specified file which it will then try to match with the program in memory. If the program has been incorrectly transferred (or it's the wrong program!), the MTX will generate a 'Mismatch' error message.

The sequence for the VERIFY operation is as follows:

1. SAVE "program"
2. Rewind tape
3. VERIFY "program"
4. Start tape
5. Press <RET>

If the MTX appears to 'lock up' in the course of VERIFYing a program, hold down the BRK key on the computer and press STOP or PLAY on the cassette recorder, and then resave the program.

RELATED KEYWORDS: SAVE, LOAD

VIEW
SYNTAX : **VIEW dir, dis**

ABBREVIATION : **VI.**

VIEW enables you to examine sections of the sprite screen that are not visible on the VDU.

The graphics screen can be thought of as a 'window' into the sprite planes. The sprite planes measure 8192 * 8192 pixels, (-4095 to +4095) whilst the screen itself is 256 * 192 pixels. Initially the screen is set so that position 0,0 on the screen is at position 0,0 on the sprite planes. The VIEW command moves the graphic screen relative to these sprite planes, in direction dir for a distance dis. dir must be in the range 0 to 7 and dis 0 to 255.

The example below places a sprite at the centre of the graphics screen, before moving the screen in every direction by a distance of 100 pixels:

```
1 REM **************
3 REM ***  VIEW  ***
5 REM **************
10 VS 4: CLS
20 CTLSPR 2,1
30 GENPAT 3,0,60,126,219,255,231,126,36,60
40 SPRITE 1,0,127,96,0,00,1
50 FOR T=0 TO 7
60 VIEW T,25
70 PAUSE 500
80 NEXT T
90 GOTO 20
```

For further information on this command refer to chapter 10.

RELATED KEYWORDS: SPRITE, ADJSPR, MVSPR

VS
SYNTAX : **VS n**

ABBREVIATION : **V.**

Used to select one of the computer's screen modes or a user-defined virtual screen.

The computer will automatically switch to the type of screen that has been selected by the statement. It is important to note that you cannot display the two text and graphics screens at the same time.

For example, VS 4 selects the full graphic screen. VS 5 selects the full text screen.

For further information on this command refer to chapter 9.

RELATED KEYWORDS: CRVS

CHAPTER 7 : SOUNDING OFF

SOUND AND THE MTX

The MTX is capable of generating an astoundingly wide range of sounds which can assault the senses through the TV speaker or your hi fi system. In other words, your MTX can make music as loud as your volume control (or ears) will permit. Not for the MTX the insipid whimpering which characterises so many micros' sound facilities - the MTX belts its sounds out in a loud positive voice!

There are no gimmicky "SPLAT" commands to give 'custard pie in the face' effects, but most sounds can be simulated using the MTX's highly flexible audio facilities. The only sound generation statement used by your micro is, unsurprisingly, the SOUND command. However, its use in conjunction with the PAUSE and SBUF statements facilitate the creation of almost any noise or musical creation.

The MTX has four sound channels, 0, 1, 2 and 3, all of which can be simultaneously activated. Channels 0, 1 and 2 are for the production of 3 note musical chords (one note from each channel), and channel 3 is a 'pink noise' generator which is used to create sound effects. Ultimately, this allows us to turn the keyboard into a full blown synthesiser with a continual drumbeat. Before we can see how this can be achieved we need to take a look at the nature of sound and how it is generated.

THE SOUND OF MUSIC

Western musical notation is organised in eight notes which are collectively known as octaves. On a piano keyboard octaves are established in sequences of seven white notes and five black notes. The C above middle C has a frequency of 512 Hz which is exactly double that of middle C, although the intervals between the notes aren't equal.

The scale of C is played entirely on the white notes from any C to the C note one octave higher (eight notes in all). The black notes, C#, D#, F#, G# and A#, are not part of the scale of C. The interval between C and D is a tone, but the interval between E and F is a semi- tone.

By playing all the notes within any octave we create what is known as a chromatic scale. For example, the chromatic scale of C consists of thirteen notes rather than the eight notes used in a normal scale. This discrepancy is a result of chromatic scales progressing in semi- tone steps.

MICRO SOUND

Sound on your micro can be generated in two ways - direct sound and continuous sound. Since the simplest of the two approaches is direct sound (which only requires three parameters), let's begin by taking a look at the most direct method of letting your micro be heard! The three parameters used by direct sound are as follows:

SOUND channel, *frequency, volume

* It is important to note that in real terms the frequency parameter is a value that is inversely proportional to frequency. Basically, the rule of thumb is the higher the value the lower the frequency.

The continuous sound command structure is more complex than direct sound as it requires the use of seven parameters. Thus:

SOUND channel, *frequency, volume, *frequency change, volume change, time, action.

Continuous SOUND statements work in conjunction with sound buffers which enables the MTX sound chip to work independently from the main processor. This allows other parts of a program to be processed while

the sound chip is busy collecting values from the sound buffers in order to generate the appropriate sounds.

THE SOUND OF SILENCE

Before we learn how persuade the MTX to produce a noise (which isn't too difficult), we should take the precaution of discovering how to turn it off (which can be tricky!). If a sound statement has been entered as a direct command, or a program finishes but the sounds don't, resultant racket can be terminated by pressing the CTRL key and G simultaneously or entering:

```
SOUND channel value,0,0
```

if the direct sound format was used, or:

```
SOUND channel value,0,0,0,0,0,0
```

if the continuous sound format was used. However, should either of these methods fail, you can resort to our final option! You can enter:

```
SBUF 1 : PRINT CHR$(7)
```

So far so good. However, if you need to silence a sound from within a program you will have to plan the termination point in advance. Having located the appropriate point in the processing you can make use of the following subroutine.

```
1000 REM SUBROUTINE TO TURN OFF SOUND
1010 SBUF 1: PRINT CHR$(7)
1020 RETURN
```

This routine will produce a short beep, after which peace and quiet will prevail. Alternatively, we could replace the CHR$ statement with: SOUND 1,0,0,0,0,0,0. Right, now we know how to stop the SOUND command let's see how we can start it!

THE SOUND OF SOUND

NOTE	FREQ (HZ)	MTX DIRECT	MTX CONTINUOUS
C	256	488	3906
C#	271	460	3687
D	287	435	3480
D#	304	411	3285
E	322	388	3100
F	342	366	2926
F#	362	345	2762
G	384	326	2607
G#	406	308	2461
A	430	290	2323
A#	456	271	2192
B	483	259	2069
C	512	244	1953

The table above lists the values required by MTX BASIC to establish the frequency of thirteen notes starting at middle C. From these values it is possible to calculate the frequency value of a given note in any octave. It should be stressed that the values in the hertz frequency column are never used in their 'raw state', but their inclusion is justified by the fact that the frequency values used by your computer are determined by calculations performed on these values.

We have already worked out the frequency values used by both direct and continuous sound and these are

listed in the MTX Direct and MTX Continuous columns. It should be stressed that these values have been rounded to the nearest integer and are not totally precise.

CALCULATING FREQUENCY VALUES

The following formula should be used when calculating the value of the frequency parameter for the direct sound command:

direct sound frequency = 125,000/Hertz value

So, to calculate the MTX direct sound frequency value for middle C you must divide 125,000 by 256. The following formula can be used when calculating the value of the frequency parameter used by the continuous sound command:

continuous sound frequency = 1,000,000/Hertz value

So 1000000/256 will return the value of the frequency parameter required to produce middle C using the continuous sound command.

The method for calculating the hertz value of a note an octave above or below the octave described in the table above is simple. You can either double the value for an octave above, or half the value for the octave below. So, to calculate the hertz frequency of the G one octave higher than the one in our chart, simply double 384 to produce a hertz frequency of 768. Similarly, to determine the frequency value of the G an octave lower, halve 384 to produce 192. Having established the hertz frequency value for the desired note, you must perform the same calculations outlined above to determine the value used by frequency parameter, be it the direct or continuous sound command.

Alternatively, you could use the table we have provided to perform the calculation by doubling the relevant frequency value for a note in the octave below or halve the value for a note in the octave above. For example, let's suppose that we wish to calculate the MTX Direct sound's frequency value for the C one octave below 'middle C'. This parameter can be achieved by simply doubling 488 (the MTX Direct's frequency value for 'middle C') and presto.

DIRECT SOUND

Having finally established how we can calculate the values of the frequency parameter, it's time we heard the SOUND command in action. Key in the following direct sound statement which uses channel 0 to produce a note:

```
SOUND 0,250,10
```

The above code uses channel 0 to generate a high-pitched note with a volume of 10 (on a scale of 0-15). Contrary to expectations the relatively low value of the statement's second parameter (250) produces a fairly high-pitched note! Remember to press CTRL and G simultaneously to stop the sound when you have had enough.

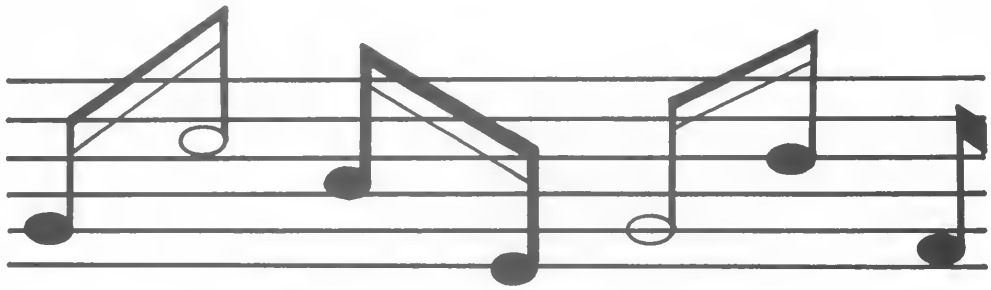
In appendix 9 we provide the formula and table which clarifies the relationship between parameter values and hertz frequencies. In this section we have established a chromatic scale for notes. By adding.

```
SOUND 1,600,10
SOUND 2,900,10
```

we can produce a chord. Since it's possible to achieve billions of combinations of three numbers between 0 and 1023. you may be wondering why certain tonal sequences are used so often. Well, as experimentation has proved. the vast majority of combinations sound awful! Sound can be a very effective form of torture as well as a spiritually uplifting experience. In other words. finding a new chord which satisfies the senses is no mean feat!

PAUSING SOUND

By now you've presumably realised that the PAUSE command is definitely self-explanatory since its sole function is to PAUSE a program. Its argument is equally straightforward: the higher its value the longer the PAUSE. However, one important point to be borne in mind when using this command is that it's not possible to gauge the exact real-time duration of a PAUSE, since the effect of the statement is dependent upon whatever other activities the MTX is undertaking when a PAUSE is executed. However, it is fairly safe to assume that PAUSE 1000 will PAUSE for approximately one second.



The chart above enables you to establish PAUSE parameters for the simulation of the temporal conventions of music (crotchets, quavers etc.). Since the objective of this chapter is to illuminate the MTX's SOUND facilites we're forced to assume that our readers possess a certain degree of musical knowledge. In other words, it's impossible to explain the meaning of a crotchet and related musical terminology without expanding the chapter into and entire book of musical theory. This said, by RUNning the programs in this chapter you should be able to glean a working understanding of the effects produced by parameter values.The following program plays a demi-semi-quaver which is gradually extended to a semi-breve:

```
5 REM  ****************************
10 REM ***   PAUSE THE SOUND   ***
15 REM ****************************
20 LET A=30
30 FOR X=1 TO 6
35 REM ********************
40 REM ***  THE NOTE  ***
45 REM ********************
50 SOUND 0,488,15
55 REM ***********************************
60 REM ***  THE LENGTH OF THE NOTE  ***
65 REM ***********************************
70 PAUSE A
75 REM ***********************************
80 REM ***  SWITCHING OFF THE NOTE  ***
85 REM ***********************************
90 SOUND 0,0,0
100 PAUSE 1000
105 REM ***********************************
```

```
110 REM ***  LENGTHENING THE NOTE  ***
115 REM ********************************
120 LET A=A*2
130 NEXT X
```

CHANNEL 3

This sound channel should be considered independently from its three companions since it produces noises rather than tones or musical notes. This is valuable when we are more interested in the creation of sound effects as opposed to music.

The following program gives an example of all the eight frequency values.

```
5 REM  ****************************
10 REM *** CHANNEL 3 EXAMPLE 1 ***
15 REM ****************************
20 FOR E=0 TO 7
30 CSR 10,10: PRINT E
40 SOUND 3,E,15
50 PAUSE 5000
60 NEXT E
```

Our example generates simulations of the following noises:

0	car horn
1	bagpipes
2	shiphorn
3	running motor
4	interference/gunfire
5	interference/gunfire
6	more interference/gunfire
7	geiger counter

It is difficult to provide a more detailed description of these noises, as, unlike visual effects, they're almost impossible to describe! The best method of finding interesting or appropriate effects is to experiment. Here is an interesting example:

```
5 REM  ****************************
10 REM *** CHANNEL 3 EXAMPLE 2 ***
15 REM ****************************
20 FOR P=1 TO 20
30 FOR V=15 TO 0 STEP -1
40 SOUND 3,4,V
50 PAUSE 300
60 NEXT V: NEXT P
70 SOUND 3,0,0
```

CONTINUOUS SOUND

The concept of a continuous sound command is more accurately described as an extended sound command. It employs more parameters than the simple direct sound command, and as a consequence is far more flexible. Continuous sound requires a total of seven parameters, where roles of the first three values correspond to those of the direct sound command (but are established by a different set of ranges). Let's refresh our memories and take another a look at the syntax employed by the continuous sound command.

SOUND channel,frequency,volume,frequency change,volume change, time, action

PARAMETERS

Channel: As with the direct sound command format, the channel parameter is used to select one of the four channels, 0, 1, 2 or 3. So there is no change to this parameter.

Frequency: The frequency parameter used by the continuous sound command structure increases in range eightfold. Thus it enables you to use values from 0 to 8192 which encompass the complete range of the MTX's potential pitch. (Although it is possible to apply values greater than 8192 to this parameter, the effect of a value is replicated according to its ratio to an end value of 8192, so the assignment of values outside this range is pointless.)

Volume: The range of this parameter must also be increased when using the continuous sound command structure. The effective range of the volume parameter is 0 to 1023. In effect, there are only sixteen levels of volume available in MTX BASIC (including 0 - silence), and by multiplying the direct sound equivalent by 64 we can assign the appropriate continuous sound amplitude.

Frequency change: This parameter is used to change the frequency (or pitch) of a note. The value of this parameter must correspond with the statements frequency parameter. Since the frequency value can be no larger than 8192, modifications to this value are only meaningful if they fall within the range of + or - 8192. Remember a plus figure gives a falling pitch and vice versa.

Volume change: Although any value in the range + or -32767 will be considered a legal parameter, the range + or - 1023 is suffice to provide maximum flexibility when using the wider volume range in the third parameter (volume).

Time: The value allocated to this parameter is measured in 64ths of a second, with a top value of 65535. In other words, the Time parameter offers a maximum duration period of 1024 seconds - over seventeen minutes! The fourth and fifth parameters (frequency change and volume change) are incremented by each unit of time. So, let's say that you have assigned a value of 1000 to the volume parameter which must be gradually turned down and, eventually, off. Under these circumstances you must set the time parameter to 1000 and the volume change parameter to -1. Thus for each unit of time the volume change parameter will deduct one from the value of the volume parameter, which will gradually turn the volume down.

Action: This parameter determines whether or not two sounds are to be linked together. Its value is either 0 or 1, where 0 links the sounds together and 1 does not. Since a sound can only be linked to the SOUND command which precedes it, the Action parameter must be set to 1 when entering the continuous sound structure as a direct command. When linking SOUND statements together there is no need to give any value but 0 to the volume and frequency parameters, since these values are established by the preceding SOUND command.

Key-in the following example and listen to the continuous sound command structure in action:

```
10 SBUF 4
20 SOUND 1,3,30,5,1,750,1
30 SOUND 0,1,0,10,0,750,1
```

The parameter values assigned can be variables or constants, or even the results of calculations returned

by keywords like SIN or COS. You may have noticed that a new command,SBUF, has been introduced in the example above. This is because more than one continuous SOUND command requires space to be allocated in the MTX Sound BUFfers in the same way as multiple values assigned to a variable require DIM statements!

SOUND BUFFERING

You might still be wondering what advantages continuous sound has over direct sound. On the face of it very similar results can be achieved using either form of sound statement by varying the frequency and volume parameters within a loop whilst using PAUSE statements to regulate the duration of a sound. Even though it is possible to achieve more or less the same effects using either command, the direct sound command structure has the disadvantage of requiring an enormous amount of programming effort to achieve its objective. More importantly, direct sound statements effectively tie up the resources of the MTX's central processor and consequently prevent it from being able to process any other instructions. However, a continuous sound command utilising the Sound BUFfers allows the CPU to process independently, while the sound chip (effectively a miniature computer in itself) collects values from the buffers and generates the appropriate audio output under its own steam!

Although the CPU pauses every 64th of a second and updates the sound chip's instructions, the speed at which the MTX operates makes this interruption virtually indetectable!

The command SBUF is used to allocate space in the MTX's Sound BUFfers. The statement takes the syntax SBUF X, where X is the number of Sound BUFfers required.

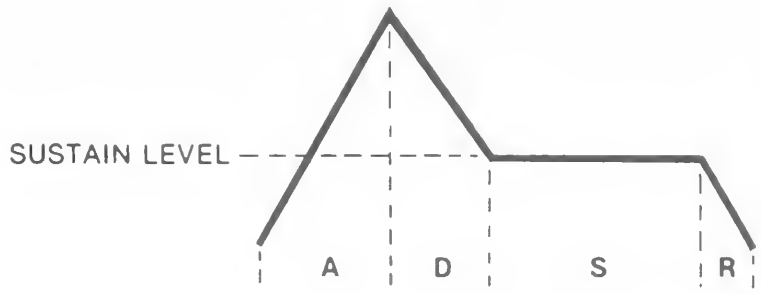
The allocation of a specified quantity of Sound BUFfers is similar to the DIMensioning of an array in that both statements reserve a block of memory for each element declared. Each sound channel uses 12 bytes per block, so if all three tone channels (0, 1 and 2) are in use, SBUF 10 requires 3*10*12 or 360 bytes.

You can declare a buffer of up to 255 blocks per sound channel, and as a rule of thumb should allocate two blocks for each continuous sound command. If you only have one continuous sound command there is no need to declare any Sound BUFfers, since the computer will automatically allocate enough memory for a single command.

As the sound chip empties one buffer, the remainder move forward one position in the queue. Since the sound is not automatically terminated the end values of the final buffer will continue unless you ensure that the ultimate volume is 0.

ENVELOPES OF SOUND

If you attempt to graphically represent the sound made by a musical instrument, the result looks something like this:



The sound represented by our diagram has four major determinants:

1	ATTACK	a sharp increase in amplitude
2	DECAY	amplitude decreasing
3	SUSTAIN	remains at a steady volume
4	RELEASE	volume fades away

Essentially a sound envelope is a measure of volume and frequency against time, and each type of musical instrument produces a radically different envelope shape.

Percussion instruments, such as drums, have rapid attacks which die very quickly, while flutes or violins have more gentle attacks and their volume is sustained for extended periods of time (consider a wind section, whose volume can only be sustained for as long as its player's breath holds out!).

The following program is an example of an envelope using the continuous sound command:

```
10 SBUF 2
20 SOUND 0,3906,900,1,-1,640,1
30 EDIT 20
```

Line 30 allows you to alter any value, but you should maintain a relatively long duration (presently 640), if you expect to assess the effect of the various parameters.

COMPUTERS AND SYNTHESISERS

A synthesiser is an electronic instrument that requires some form of keyboard input to produce the notes required. The different sounds (and instruments) generated by the synthesiser are simulated by voltage controlled electronic oscillators. The tone produced by these oscillators is further modified by voltage controlled filters and envelope generators, giving sounds such as vibrato (warbling) and lengthy sustains.

Computers can be programmed to behave as crude synthesisers by transforming user-generated digital input into its analogue equivalent. Such digital input can be stored in the micro's sound buffers until it reaches the front of the queue when it can be output as sound. (The electric piano program graphically demonstrates this principle.)

ADVANCED MUSIC

As the frequency of upper C is double that of middle C we can easily calculate the frequency change for each interval. (For the mathemeticians amongst you, the formula used is 12^2 (the twelfth root of two). However for our purposes, 1.0594631 is a more convenient representation.) Since the hertz frequency of middle C is 256, we can calculate the hertz frequency of C# by multiplying 256 by 1.0594631. Similarly, to get the frequency of D we simply multiply the frequency of C# by the same number.

As we have already seen, the main point to be borne in mind when applying the results of such calculations is that the MTX uses the hertz frequency to define the pitch of a note. This consideration is further complicated by the fact that there are two versions of the SOUND command whose parameters utilise different sets of ranges.
Let's quickly recap on the relationships between the hertz frequency and the MTX's frequency paramaters. Divide a given number (let's call it X) by the hertz frequency and the MTX parameter is produced. In BASIC, MTX frequency = X/hertz frequency.When generating direct sound X is equal to 125,000, while 1,000,000 must be used in the calculation of the continuous sound frequency parameter.

To obtain the values of notes in other octaves, the MTX parameter must be halved for each higher octave, and doubled for each octave which is lower. Note that the maximum parameter for direct sound is 1023 and 8192 for continuous sound.

CHAPTER 8 : GRAPHICS WITH CHARACTER

Your MTX possesses a formidable armoury of graphics capabilities which facilitate both practical and aesthetic application. In the first chapter of our graphics section we shall examine the techniques used in the manipulation of the text screen. The following chapter will enable you to come to grips with the MTX's High Resolution graphics screen features, which leads on to a final chapter which examines the use of sprites on the MTX.

THE TEXT SCREEN

When you first turn on the MTX you are confronted with the Text Screen and whenever a program has finished RUNning the computer will always return to the text mode. The selection of text or graphics mode is established by the coding of your program. As you'll discover, MTX BASIC also offers the possibility of creating several 'virtual screens' within the larger main screen. But let's unveil the MTX's graphics potential one step at a time. For the moment we'll concentrate on a single text screen.

Whenever a program is RUN the MTX automatically selects the text screen unless otherwise instructed. There may be times when you will want to switch between different screens within your programs, so if you want to select the text screen you will need to use the command VS 5. For instance:

```
10 VS 5: CLS
```

selects the full text screen. We will be including this command in most of the programs in this section. It should be stressed, however, that this statement is not always essential as the MTX automatically selects the text screen when no other is specified. Our example's second command, CLS, CLears the Screen of any stray characters and renders it ready for use. CLS can be used in both text and graphics modes whenever you wish to clear a screen display. Having programmed a fresh start for ourselves, the next thing to do is to PRINT something. Add the following lines:-

```
50 PRINT "HELLO"
200 GOTO 200
210 REM*** PRESS BRK TO EXIT ***
```

When you RUN the modified code, the word HELLO should appear in the top left corner of the screen. Not very exciting, but it's a start. Let's change the background colour of the screen by using the PAPER command, whose argument determines the number of the colour required. While we're about it, we might as well change the colour of the text. This can be achieved using the INK command, which once again uses a colour code to define the pigment. Add the following line and the "HELLO" will be printed in black upon a yellow background:-

```
20 PAPER10: INK 1
```

Our next example displays all of the combinations of PAPER and INK colours:-

```
1 REM *** INK AND PAPER EXAMPLE ***
10 VS 5: CLS
20 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 FOR PAP=0 TO 15
40 PAPER PAP
```

```
50 FOR PEN=0 TO 15
60 INK PEN
70 PAUSE 500
80 NEXT PEN
90 NEXT PAP
```

The main point of interest in this program is that not all of the colour combinations are readable. When designing your own programs you must be careful to choose a suitable combination of PAPER and INK colours. Also you should notice that the INK command changes the colour of all of the text on the screen, since you can only have a maximum of two colours on the text screen - the text (INK) colour and the screen (PAPER) colour.

Having added some colour to our display, our next task is to arrange our text in a more orderly fashion in relation to the screen. After all, printing on the top line is rather boring and restrictive, isn't it? The MTX possesses two methods of formatting text, the PRINT statement and the CSR command.

Type in and RUN the next example. (You should, of course, type NEW and then press the RET key to get rid of any other program that might be in memory before starting.)

```
1 REM **************
3 REM *** PRINT ***
5 REM **************
10 VS 5: CLS
20 PRINT "HELLO, ": PRINT "MY NAME IS "
30 PRINT "FRED "
200 GOTO 200
210 REM *** PRESS BRK TO EXIT PROGRAM ***
```

On RUNning the program you should see three statements PRINTed on three separate lines. The computer will always PRINT on a new line whenever it comes across a PRINT statement. However, there are some special PRINT formats which alter the effects of a statement. Type in an alternative line 20. Only the two commas are new so you can use the EDIT facilities to insert them into your existing lines if you wish:

```
20 PRINT "HELLO ",: PRINT "MY NAME IS ",
```

RUN the program again and you should see that the PRINT statements generate an alternative display. The MTX text screen can be thought of as having five columns (or format-fields), each eight characters wide. A comma in a PRINT statement tells the computer to start the next PRINT instruction from the beginning of the next field or column, rather than the next line. Add line 40 to convince yourself that the comma only affects the PRINT statement in which it appears:

```
40 PRINT "A"
```

The 'A' appears on a new line because there is no comma ending the previous PRINT statement on line 30. Add a comma to the end of line 30 and take note of the result.

A second print separator, as these characters are known, is the semi-colon. A semi-colon causes the PRINT position to remain wherever it was after the last PRINT statement was executed. Changing the commas in your program to semi-colons will highlight a hidden ability of the your micro. The program should now look like this:-

```
10 VS 5: CLS
20 PRINT "HELLO ";: PRINT "MY NAME IS ";
30 PRINT "FRED";
40 PRINT "A"
200 GOTO 200
210 REM*** PRESS BRK TO EXIT ***
```

The output is now arranged along the top line. What is the hidden ability? The fastest sex-change operation in the world! When PRINTing numbers the computer will always PRINT a space in front of positive numbers, instead of the + sign. This space is there to allow for the minus sign of negative numbers so that numbers can be easily lined up in columns regardless of their sign. Before we move on to the CSR command, add these lines to FREDA'S program to demonstrate the format when using numbers:-

```
50 PRINT "A","B","C","D","E"
60 PRINT 1,2,3,4,5
70 PRINT -1,-2,-3,-4,-5
```

For completeness, change all of the commas in lines 50, 60, and 70 to semi-colons and observe the changes to the output when you re-RUN the program.

Using the PRINT command on its own to form screen displays is fairly limited in its usefulness, even when using print separators. However, MTX BASIC provides us with the CSR command which enables text to be positioned anywhere on the screen. The format of this statement is CSR x,y; this positions the cursor at column x on row y ready to start PRINTing. The next following PRINT statement will start from position x,y.

The full MTX text screen facilitates 24 lines of text, each of which comprises 40 character positions or columns (a total of 960 positions). The lines are numbered 0-23 and the columns run from 0 to 39, where the top left corner of the screen is position 0,0. Using CSR x,y you can position the CurSoR anywhere on the screen. If you have trouble remembering which direction is x and which is y then try to think of this aide-memoire: X is a cross so wise up (y is up)!

The next example demonstrates how a character can be moved around the screen using CSR and PRINT statements. This method of animation can be quite effective when used as a feature of your own games programs. Later we'll unravel the mysteries of creating user-defined characters, but for now you'll have to be content with moving a '*' symbol around.

```
10 REM MOVEMENT DEMONSTRATION
20 VS 5: CLS
30 REM *** DEFINING VARIABLES ***
40 LET X=0: LET Y=0: LET X1=0: LET Y1=0
50 REM *** SETTING UP KEYS ***
60 REM *** SETTING UP MOVEMENT ***
70 IF INKEY$="A" AND Y>0 THEN  LET Y=Y-1
80 IF INKEY$="Z" AND Y<23 THEN  LET Y=Y+1
90 IF INKEY$="N" AND X>0 THEN  LET X=X-1
100 IF INKEY$="M" AND X<38 THEN  LET X=X+1
110 IF X<>X1 OR Y<>Y1 THEN  CSR X1,Y1: PRINT " ";
120 CSR X,Y: PRINT "*";
130 LET X1=X: LET Y1=Y
```

```
140 GOTO 50
```

When you RUN this program you'll be able to move the asterisk (*) around the screen using the keys A-up, Z-down, N-left and M-right. This routine can be used as the basis for many types of games programs, so it's worth examining in detail.

Line 40 sets up the variables X and Y. Lines 70 to 110 contain the routine that moves the shape in the direction determined by the key being pressed. Note that all the statements which animate our asterisk are virtually identical in format. Line 40 first checks whether the 'A' key is being pressed, if it is it then checks the value of Y. As long as the variable Y is greater than 0 a value of one will be deducted from the total value which will cause the asterisk to move up the screen. In plain English, line 70 says "IF you want to move up AND you are not at the top of the screen THEN move up one line." The actual movement takes place in line 120, using CSR to position the CurSoR so for the time being the variable Y is altered.

The other three lines (80, 90 and 100) perform similar tasks for the other three directions and then the computer reaches line 110. This line checks whether a movement is requested. If all of the values of X, Y and X1, Y1 are the same then the shape will not move. If, however, any one co-ordinate has changed then the shape will move and we must first of all remove the shape from its old position. This is done by printing a space at the old co-ordinates. Line 120 then prints the shape at the new position. In line 130 the old position is updated to the current position before, in line 140, the computer repeats the whole process.

There are several BASIC commands that allow you to manipulate string variables in many ways. LEFT$, RIGHT$, VAL, STR$ and so on are all covered elsewhere and will not be dwelt upon in this chapter. There are two BASIC functions that are important to us at present however, CHR$ and SPK$.

The function CHR$(x) is used to obtain any character within the (extended) ASCII code. The ASCII code is a standardised representation of the characters used by the majority of computers, and for a complete list of these codes see Appendix 4. Although a number of these characters are not directly printable, CHR$(x) can be used to access the majority of the set. For example entering PRINT CHR$(65) tells the computer to PRINT an A on the screen, whilst PRINT CHR$(8) will PRINT a 'back-space' (move the cursor back one position).

SPK$ is a function unique to MTX BASIC and can be extremely useful in the generation of screen displays. SPK$ is the Screen PeeK function which returns the character at the current cursor position, then moves the cursor to the next PRINT position. There are no parameters associated with SPK$, so the cursor must be placed at the required position using CSR x,y before SPK$ can be used to read the character at that position. The next program is an enhanced version of our earlier 'star mover' example. The subroutine starting at line 500 puts obstacles on the screen and line 85 uses SPK$ to prevent you from bumping into the obstacles. Type NEW, as usual, before keying in this program:

```
10 REM MOVEMENT WITH OBSTACLES
20 VS 5: CLS
30 LET X=0: LET Y=0: LET X1=0: LET Y1=0
35 GOSUB 500
40 IF INKEY$="A" AND Y>0 THEN  LET Y=Y-1
50 IF INKEY$="Z" AND Y<23 THEN  LET Y=Y+1
60 IF INKEY$="N" AND X>0 THEN  LET X=X-1
70 IF INKEY$="M" AND X<38 THEN  LET X=X+1
80 IF X<>X1 OR Y<>Y1 THEN  CSR X1,Y1: PRINT " ";
85 CSR X,Y: IF SPK$<>CHR$(32) THEN  LET X=X1: LET
Y=Y1
90 CSR X,Y: PRINT "*";
100 LET X1=X: LET Y1=Y
```

```
120 GOTO 40
500 REM RANDOM OBSTACLES
510 FOR T=1 TO 15
520 CSR RND*38,RND*23
530 PRINT CHR$(65+RND*26)
540 NEXT T
550 RETURN
```

Line 85 prevents the 'star' from running over any of the randomly placed objects. First of all, the CSR X,Y instruction moves the cursor (which is invisible, of course) to the new position X,Y. SPK$ checks that there is a space, CHR$(32), at that position. If there is anything other than a space then the X,Y position is reset to the old position X1,Y1 so that, in effect, no movement takes place. If there is a space then movement is permitted. The actual movement is still carried out in line 90 just as in the earlier example.

This program seems to be working towards a game, doesn't it? If you constructed a maze instead of the random obstacles and if you had a 'ghost' or monster chasing you through the maze while you eat all the dots that are scattered around you would have a fair rendition of a 'PAC-MAN' type game. What do we need next? For now we'll leave the construction of the maze to you. Why not put the manual down for a while and see if you can constuct the maze by yourself?. Don't worry if you can't because we'll be coming back to this program later in the chapter and adding a routine that draws a maze for you.

As we have discovered, a PRINT statement enables characters to be PRINTed to the screen. But where do these characters come from? Inside the computer there is an area in ROM in which the MTX stores all the information required to make up each and every character. Each character is formed in a grid of 8 dots across by 8 dots down.

Thus character grids comprise 64 dots (i.e. eight dots down and eight across). The dots that go to make up this grid are called BITs (which is short for Binary digIT) and as there are eight bits to a byte the lines across this character grid can be stored in a single byte. If a dot (or pixel as it is properly known) is 'on' then the bit is set to a 1, if the pixel is 'off' then the bit is set to 0.

USER DEFINED CHARACTERS

The MTX makes the process of designing characters particularly easy as there is a special BASIC command, GENPAT which is used to GENerate PATterns. By using GENPAT you can create your own charactes and sprites. Since sprite creation is covered in chapter 10 we will concentrate on the creation of character shapes in this chapter. The format for the GENPAT command is:

GENPAT p,n,d1,d2,d3,d4,d5,d6,d7,d8

It looks formidable doesn't it? However, despite appearances to the contrary it is quite simple to use. Starting from the end of our syntax example, the numbers d1 to d8 are the eight items of data that are used to define the shape of the character. As we have just seen, the data for the A shape is 16,40,68,68,124,68,68,0. When you create your own character shape you add up the bit pattern for each row and use the resulting eight numbers, starting with the top row, in the GENPAT command to define your own shape.

As previously stated, the GENPAT command is used to define sprites as well as user-defined characters. The parameter p is used to tell the computer which type of shape we wish to create and this parameter requires values in the range 0-7, which determine the type of the shape. A full explanation of the values 3-7 will be provided in chapter 10, so for the moment we shall concentrate only on the p values 0 to 2. Setting p=0 allows you to redefine one of the ASCII characters from code 32 to code 127. Any, or all, of these characters can be redefined using successive GENPAT statements. The second parameter, n, is the code of the particular character that is to be redefined and so must be in the range 32 to 127.

Setting p=1 allows you to redefine one of the non-ASCII characters, whose codes are between 129 and 154. These 25 characters cannot be typed in directly from the keyboard but can be PRINTed onto the screen using the CHR$(x) function. They can be used in any situation where you need some user-defined characters as well as the full standard character set. If p=1 the value for n must be between 129 and 154 to specify the particular character for redefinition.

Setting p=2 allows you to create up to 8 multi-coloured user-defined characters where the characters have the ASCII codes 147 to 154. Having previously defined the shape of one of these characters, using p=1, you can then define the INK and PAPER colour for each individual row of the character. When using p=2, each of the numbers d1 to d8 is used to define the colours for a single horizontal row rather than the dot-pattern of the shape. The data values are made up using the colour numbers (0-15) where the total for each row is:

VALUE= 16 * PAPER + INK

For example, you wanted the top row of your character to have a black PAPER colour and a light yellow INK colour then the value for d1 would be:

VALUE(d1) = 16 * 1 + 11 = 27

You can only have two colours on each row, one PAPER colour and one INK colour, but you can define different combinations of colours for each of the eight rows. You can, therefore, have up to 16 colours within a single character, a graphics facility unique to the MTX range. Most other home computers struggle to support four colours!

When creating your own character, the first step is to design the shape on an 8*8 grid like the one below which illustrates the C character. If your character is going to be used on a text screen you can't define the bit 0 and bit 1 columns. This is because the characters on the text screen are only six pixels wide as opposed to the full 8*8 character that can be displayed upon a high-resolution graphics screen. When you are happy with your shape you must add up the totals for each row to obtain eight numbers for the GENPAT statement. Finally, you must select the character you are going to redefine and allocate the values of p and n accordingly. For example:

VALUES									CALCULATION		DATA
	1										
	2	6	3	1							
	8	4	2	6	8	4	2	1			
		*	*	*	*				64+32+16+8	=	120
	*	*	*	*	*	*			128+64+32+16+8+4	=	252
	*	*	*						128+64+32	=	224
	*	*							128+64	=	192
	*	*	*						128+64+32	=	224
	*	*	*	*	*	*			128+64+32+16+8+4	=	252
	*	*	*	*	*	*			128+64+32+16+8+4	=	252
		*	*	*	*				64+32+16+8	=	120
BIT	7	6	5	4	3	2	1	0			

Right, now we have a PAC-MAN shape, let's replace the asterisk (*) character with it.

GENPAT 0,42,120,252,224,192,224,252,252,120

This can be typed in directly, as above, or you can add a line number and use the command within a program. If you type it in directly (and press RET) then, whenever you type the '*' on the keyboard the PAC-MAN shape will be PRINTed instead.

Adding the following line to our movement demonstration program will allow you to move the new shape around instead of the 'asterisk'.

```
15 GENPAT 0,42,120,252,224,192,224,252,252,120
```

So, the first GENPAT paramater (p) tells the MTX that you wish to replace a character with an ASCII code between 32 and 127 and the second paramater (n) tells it that the character you wish to replace is the asterisk (*) as 42 is its ASCII code. If you key-in the movement program again with this additional line and then RUN the program, you will see that the PAC-MAN is always facing to the right no matter what direction he is moving, which is not very realistic, is it?

SIMPLE ANIMATION USING CHARACTER GRAPHICS

Simple animation can be achieved quite effectively on the text screen by creating several user-defined characters and then PRINTing them alternately on the screen. In the case of our solitary PAC-MAN we need to have a total of 5 shapes. We need four 'open-mouthed' shapes for each direction of movement and a single, general purpose 'closed mouth' shape. The shape required for downward movement is as follows:

* * * *	64+32+16+8	= 120
* * * * * *	128+64+32+16+8+4	= 252
* * * * * *	128+64+32+16+8+4	= 252
* * * *	128+64 +8+4	= 204
* * * *	128+64 +8+4	= 204
* *	128 +4	= 132
* *	128 +4	= 132
		= 0

and so the GENPAT statement for our downward PAC-MAN is:

```
16 GENPAT 0,43,120,252,252,204,204,132,132,0
```

Which redefines the character with an ASCII code of 43, the plus sign. Similarly, the upward facing shape is defined by:

```
17 GENPAT 0,44,0,132,132,204,204,252,252,120
```

and the left facing shape is defined by the following GENPAT statement:

```
18 GENPAT 0,45,120,252,28,12,28,252,252,120
```

The final shape is the 'closed-mouth' shape, which is defined by:

```
19 GENPAT 0,46,120,252,252,252,252,252,252,120
```

If you try to RUN the program you will notice that our newly defined characters aren't actually being PRINTed to the screen. This is because the asterisk is the ony character that is being PRINTed. Why don't you see if you can get all of our PAC-MEN moving round the screen.

The following program listing is the completed PAC-MAN type game, using the shapes described above and a similar method of moving around the screen as we used in the 'star mover' routines earlier in the chapter. Points of special interest in this demonstration are, firstly the method for obtaining the simple animation, lines 40 to 110. The variable A$ stores the relevant 'open-mouth' shape for the direction of travel. This is not directly PRINTed onto the screen. Instead line 87 increments a counter on each pass through this section of the program between the values 0 and 4.

Line 89 then compares the COUNT. IF the value of COUNT is greater than 2 THEN B$ is set equal to the 'closed-mouth' shape, CHR$(46), ELSE B$ is set equal to the current 'open-mouth' shape A$. B$ is then PRINTed onto the screen in line 90. The second interesting point is the simplicity of drawing a maze shape using MTX BASIC. Twenty-three program lines, each of which PRINTs one screen line using Q for the walls and a ' (SHIFT and 7) for the pills for the PAC-MAN to eat. You can create your own maze pattern by altering these lines, 500 onwards.

```
10 REM PAC-MAN DEMO
15 GENPAT 0,42,120,252,224,192,224,252,252,120
16 GENPAT 0,43,120,252,252,204,204,132,132,0
17 GENPAT 0,44,0,132,132,204,204,252,252,120
18 GENPAT 0,45,120,252,28,12,28,252,252,120
19 GENPAT 0,46,120,252,252,252,252,252,252,120
20 GENPAT 0,36,48,120,180,252,252,252,252,168
25 VS 5: CLS
30 LET X=1: LET Y=2: LET X1=0: LET Y1=0: LET A$=CH
R$(42): LET B$=CHR$(46): LET COUNT=0: LET SK$=""
32 LET SCORE=0: CLOCK "000000"
34 LET G1X=35: LET G1X1=35: LET G1Y=2: LET G1Y1=2
35 GOSUB 500
40 IF INKEY$="A" AND Y>0 THEN  LET Y=Y-1: LET A$=C
HR$(44)
50 IF INKEY$="Z" AND Y<23 THEN  LET Y=Y+1: LET A$=
CHR$(43)
60 IF INKEY$="N" AND X>0 THEN  LET X=X-1: LET A$=C
HR$(45)
70 IF INKEY$="M" AND X<38 THEN  LET X=X+1: LET A$=
CHR$(42)
80 IF X<>X1 OR Y<>Y1 THEN  CSR X1,Y1: PRINT " ";
85 CSR X,Y: LET SK$=SPK$: IF SK$<>CHR$(32) AND SK$
<>CHR$(39) THEN  LET X=X1: LET Y=Y1
86 IF SK$=CHR$(39) THEN  LET SCORE=SCORE+1: CSR 15
,0: PRINT "SCORE=";SCORE
87 LET COUNT=COUNT+1: IF COUNT>4 THEN  LET COUNT=0
89 IF COUNT>2 THEN  LET B$=CHR$(46) ELSE  LET B$=A
$
90 CSR X,Y: PRINT B$;
100 LET X1=X: LET Y1=Y
105 GOSUB 200
110 GOTO 40
200 REM MOVE GHOST 1
```

```
210 IF G1X>X THEN  LET G1X=G1X-1 ELSE  IF G1X<X TH
EN  LET G1X=G1X+1

215 CSR G1X,G1Y: LET G1$=SPK$: IF G1$<>CHR$(32) AN
D G1$<>CHR$(39) THEN  LET G1X=G1X1 ELSE  CSR G1X1,
G1Y1: PRINT " ";: CSR G1X,G1Y1: PRINT "$";

217 IF G1$>CHR$(41) AND G1$<CHR$(47) THEN  FOR T=0

 TO 15: PAPER T: PAUSE 100: NEXT T: GOTO 1000

220 IF G1Y>Y THEN  LET G1Y=G1Y-1 ELSE  IF G1Y<Y TH
EN  LET G1Y=G1Y+1

230 CSR G1X,G1Y: LET G1$=SPK$: IF G1$<>CHR$(32) AN
D G1$<>CHR$(39) THEN  LET G1Y=G1Y1 ELSE  CSR G1X,G
1Y1: PRINT " ";: CSR G1X,G1Y: PRINT "$";

260 LET G1X1=G1X: LET G1Y1=G1Y

280 IF G1$>CHR$(41) AND G1$<CHR$(47) THEN  FOR T=0
 TO 15: PAPER T: PAUSE 100: NEXT T: GOTO 1000

290 RETURN

500 REM PRINT THE MAZE

502 GENPAT 0,81,252,252,252,252,252,252,252,252

504 GENPAT 0,39,0,0,0,48,48,0,0,0

510 CSR 0,1: PRINT "QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
QQQQQQQQQ";

520 CSR 0,2: PRINT "Q'''''''''''''''''''''''''''''
'''''''''Q";

530 CSR 0,3: PRINT "Q'QQQQQQQQQQ'QQQQQQQQQQQQQ'QQQ
QQQQQQQ'Q";

540 CSR 0,4: PRINT "Q'QQQQQQQQQQ'QQQQQQQQQQQQQ'QQQ
QQQQQQQ'Q";

550 CSR 0,5: PRINT "Q'QQ''''''''''''''''''''''''''
'''''QQ'Q";

560 CSR 0,6: PRINT "Q'QQ'QQQQ'QQQQ'QQQQQQQQQ'QQQQ'
QQQQ'QQ'Q";

570 CSR 0,7: PRINT "Q''''QQQQ'QQQQ''''''''''QQQQ'
QQQQ''''Q";

580 CSR 0,8: PRINT "QQQQ'QQQQ'QQ'''QQQQ'QQQQ'''QQ'
QQQQ'QQQQ";

590 CSR 0,9: PRINT "QQQQ'QQQQ''''Q'Q'''''''Q'Q''''
QQQQ'QQQQ";

600 CSR 0,10: PRINT "Q'''''''QQQQ'Q'Q'QQQQQ'Q'Q'QQ
QQ'''''''Q";

610 CSR 0,11: PRINT "Q'QQQQQ'QQQQ'Q'Q'''''''Q'Q'QQ
QQ'QQQQQ'Q";

620 CSR 0,12: PRINT "Q'QQQQQ'''''''Q'QQQQQ'Q'''''
'''QQQQQ'Q";

630 CSR 0,13: PRINT "Q'QQQQQ'QQQQ'Q'Q'''''''Q'Q'QQ
QQ'QQQQQ'Q";

640 CSR 0,14: PRINT "Q'''''''QQQQ'Q'Q'QQQQQ'Q'Q'QQ
QQ'''''''Q";

650 CSR 0,15: PRINT "QQQQ'QQQQ''''Q'Q'''''''Q'Q'''
'QQQQ'QQQQ";

660 CSR 0,16: PRINT "QQQQ'QQQQ'QQ'''QQQQ'QQQQ'''QQ
'QQQQ'QQQQ";

670 CSR 0,17: PRINT "Q''''QQQQ'QQQQ''''''''''QQQQ
'QQQQ'''Q";

680 CSR 0,18: PRINT "Q'QQ'QQQQ'QQQQ'QQQQQQQQQ'QQQQ
'QQQQ'QQ'Q";

690 CSR 0,19: PRINT "Q'QQ''''''''''''''''''''''''
''''''QQ'Q";

700 CSR 0,20: PRINT "Q'QQQQQQQQQQ'QQQQQQQQQQQQQ'QQ
QQQQQQQQ'Q";
```

```
710 CSR 0,21: PRINT "Q'QQQQQQQQQQ'QQQQQQQQQQQQQ'QQ
QQQQQQQQ'Q";

720 CSR 0,22: PRINT "Q''''''''''''''''''''''''''''
''''''''''Q";

730 CSR 0,23: PRINT "QQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
QQQQQQQQQQ";

800 RETURN

1000 REM END OF GAME

1010 VS 5: CLS

1020 CSR 4,3: PRINT "YOUR SCORE WAS ";SCORE

1030 CSR 4,5: PRINT "YOU LASTED ";

1040 PRINT MID$(TIME$,3,2);" MIN ";MID$(TIME$,5,2)
;" SECS"
```

REFORMED CHARACTERS

There is another interesting and simple method of creating animated displays with user-defined character graphics: By using the GENPAT command to change characters that are already on the screen, the changes are made in real-time!

In the example which follows a block of A's are initially PRINTed in the middle of the screen. The tricks start at line 100 where, within a FOR...NEXT loop, the PATtern of the A is redefined. Type it in and give it a try. Remember that this is only a demonstration and you can change it around any way you wish. Using multi-coloured characters can be particularly stunning and don't forget that you can change several different character patterns.

```
10 REM ************************
12 REM * DEMONSTRATION PROGRAM *
14 REM *   CHANGE CHARACTERS   *
16 REM *  THAT ARE ALREADY ON  *
18 REM *       THE SCREEN      *
20 REM ************************
30 FOR T=3 TO 13
40 CSR 10,T: PRINT "AAAAAAAAAAAAAAA";
45 PAUSE 200
50 NEXT T
100 FOR X=2 TO 7
110 LET Y=2^X
120 GENPAT 0,65,Y,Y,Y,Y,Y,Y,Y,Y
130 PAUSE 200
140 NEXT X
150 GOTO 100
```

After you've RUN this short program you may not be over impressed by the results. Remember that it is a demonstration program, you can write something better, can't you? When you have read through the next chapter on high-resolution graphics, you can then come back to this program and write an example that (tries) to perform the same task using the hi-res commands. In so doing you'll appreciate the speed of the techniques we have just outlined.

CHAPTER 9 : HI RESOLUTION GRAPHICS

SCREENS

In the pre-ceding chapter we took a fairly extensive look at the MTX's Text Screen. In this chapter we will be exploring the capabilities and potential of the full Graphics Screen - the so-called 'High-Resolution Screen'. As you probably know, the MTX's text screen is also known as the low-resolution screen, but before we go any further, it is probably worth taking some time out to clarify the concept of screen 'resolution'.

The full-size text screen allows us to place characters in forty columns across the screen and in twenty-four lines down the screen (or up, if you're standing on your head!). You cannot, however, place a character BETWEEN two column positions, nor BETWEEN two lines of text. The text screen should be considered as a grid of 960 squares (40*24=960); thus, it is possible to place characters in a maximum of 960 positions on the screen. The resolution of this type of screen is 960 squares, which is considered a low-resolution screen.

We have already seen how each character displayed on the text screen comprises a series of dots in an eight by six dot grid. We also established that it is possible to alter the dot (or pixel) pattern of any character by creating 'user-defined' characters. Thus it should be fairly obvious to the alert reader that the screen is actually made up of many more pixels (short for Picture Elements) than the 960 character positions that the text screen allows you to use.

Once the potential of this revelation has finally hit home, the graphics enthusiast will be relieved to learn that it is possible (with a large amount of programming effort) to individually control these pixels. For instance, by turning certain pixels on and others off we can produce a circle on the screen! In fact, the MTX graphics screen and its associated commands offer a potential far in advance of the majority of micros in your micro's price range.

The full graphics screen is made up of 256 pixels across the screen and 192 pixels running vertically down the screen. This gives us a total of 49,152 individually addressable pixels. Since it is possible to turn individual pixels on or off (all 49,152 of them!) the full graphics screen is described as a 'high resolution' screen. Later on in the chapter, you will see that the screen colour resolution is 6144 locations (which, presumably could be described as as medium- resolution). But enough of theory! Let's see how we can exploit the MTX's graphics facilities.

The full graphics screen is accessed by using the command:

VS 4

This should always be followed by the CLS command, which clears the screen to the current background colour. The first important point to remember is that while it is possible to put text on to the graphics screen, it isn't possible to draw graphics on the text screen. You'll be glad to learn that it is just as simple to PRINT text on the hi-res screen as it is to PRINT to the text screen. In fact, the hi-res screen accomodates the majority of the text-handling functions that we saw in the last chapter.

This said, it is important to note that the size of individual characters on the hi-res screen is eight by eight pixels and not six by eight pixels of the text screen. Since there are 256 pixels across the graphic screen, the screen size for text becomes 32 characters (256/8) across, by 24 lines down and the characters are now two pixels wider. When the standard character set is being used, the extra width of each character widens the gap between each letter or number. When creating user-defined characters for the high-resolution screen you must design your shapes using the full eight by eight grid or the resultant shapes will look rather lopsided! You can see this quite clearly by changing line 20 of the PAC-MEM program in the last chapter. Thus:

```
20 VS 4: CLS
```

When you RUN the program you should notice three things of particular interest.

1. The computer makes a laudible attempt at displaying the maze. Most home micros would have floundered if they had been asked to shift from a text screen to a graphics screen. But not your MTX! The maze from our PAC-MEM program has not retained its original shape because the parameters used by the CSR command on the text screen are different to those which are applied to the graphics screen. On the hi-res screen there are only 32 columns and consequently the x- parameter for CSR statement must be within the range 0-31.

2. User-defined characters are still defined and manipulated in exactly the same way (however, you should use the eight by eight grid when creating characters on the graphics screen). Similarly, all of the string handling functions such as CHR$, LEFT$, RIGHT$ and so on are unchanged. In essence, text is handled on the graphics screen in exactly the same manner as on the text screen, providing you allow for the shorter line length.

The only commands that don't perform in exactly the same way on both screens are PAPER and INK. But never fear - the MTX is equipped with the COLOUR command which provides access to the same facilities. The MTX is unique in the home computer world as it permits a diverse mixture of graphical modes to co-exist with very little programming effort.

3. The final point to notice as far as PAC-MEM is concerned is that the maze now displays blocks of vertical stripes. These are caused by the wider characters used on the hi-res screen. The walls of the Text screen maze were defined using a 6*8 character and the computer is now displaying an eight by eight character with the first two pixels of each character row set to a zero. As previously explained, when creating user-defined characters on the graphics screen you should remember to use eight by eight characters.

The MTX possesses a suite of customised BASIC commands and functions for use on the graphics screen. The first we shall outline is the aforementioned COLOUR command, which takes the following format:

COLOUR p,n

where p is the parameter which establishes the display element whose pigment is to be defined and n is the number assigned to the colour. The parameter p uses values in the range 0-4 to effect the following screen criteria:

value of p	function
0	PRINT BACKGROUND COLOUR
1	PRINT INK COLOUR
2	PLOTTING BACKGROUND COLOUR
3	PLOTTING INK COLOUR
4	SCREEN BORDER COLOUR

The value of n can be any of the fifteen colour codes supported by MTX BASIC, or when set to zero can specify transparent displays. (In other words, a zero allocation sets the foreground display colour to the current background colour.) Unlike the PAPER and INK used on the text screen, a COLOUR command with parameter p=0 or p=1 will only affect PRINT statements following the COLOUR command - it does not effect the colour of text that is already on the screen. The following program will mutate the screen through all fifteen colours:

```
10 REM **********************
12 REM *   GRAPHIC SCREEN   *
14 REM *   DEMO PROGRAM 1   *
16 REM **********************
20 VS 4: CLS
30 COLOUR 0,1: REM BLACK PAPER
40 COLOUR 1,5: REM LIGHT BLUE INK FOR PRINT
```

```
50 COLOUR 3,1: REM BLACK PLOT COLOUR
60 FOR T=1 TO 15
70 COLOUR 2,T: CLS : PAUSE 500
80 NEXT T
90 GOTO 90
100 REM *** PRESS BRK TO EXIT PROGRAM ***
```

The most important point demonstrated by the above routine is the inclusion of the CLS command in line 70. You must clear the graphics screen before you attempt to change its background colour. It is not possible to globally change the colour of the screen whilst maintaining the current screen display. Remove the CLS instruction from line 70 and you will find that no colour changes take place. If the CLS instruction is omitted, the COLOUR command will only change the background colour for the next plotting command. Change line 70 and add line 75 to see the difference:

```
10 REM **********************
12 REM *   GRAPHIC SCREEN    *
14 REM *   DEMO PROGRAM 1    *
16 REM **********************
20 VS 4: CLS
30 COLOUR 0,1: REM BLACK PAPER
40 COLOUR 1,5: REM LIGHT BLUE INK FOR PRINT
50 COLOUR 3,1: REM BLACK PLOT COLOUR
60 FOR T=1 TO 15
70 COLOUR 2,T: PAUSE 500
75 PLOT RND*255,RND*191
80 NEXT T
90 GOTO 90
100 REM *** PRESS BRK TO EXIT PROGRAM ***
```

Now we have fifteen points PLOTted on the screen, each with a different background colour (of course you can only see fourteen points because one of them is the same colour as the screen). Since line 75 PLOTs these points at RaNDom positions around the screen the display will be different each time you RUN the program.

Earlier we mentioned that the colour resolution on the graphics screen is not as great as the PLOTting resolution. There are 49,152 individual pixels on the screen but only 6144 (49152/8=6144) separate colour locations. Each location is eight pixels wide by one pixel deep, which will only permit one PLOTting colour and one background colour within a single location. This is why our example displays fourteen horizontal bars (the background colour), each containing a single PLOTted point. If you examine the display carefully you'll discover a single point by itself, this is because the background colour for that point is the same as the background colour of the screen.

Although the colour resolution on your micro is considerably more sophisticated than on most home computers, there are invariably occasions when compromise is the order of the day. For instance, drawing two intersecting lines, where each line uses different background and PLOTting colours, will cause a colour abberration at the point at which the lines cross. This only becomes unacceptably conspicuous when vertical or near vertical lines are used.

PLOTTING

PLOT is a simple command which allows you to activate individual pixels using the syntax:

PLOT x,y

where x and y are the co-ordinates of the point you want to PLOT to the screen. The x co-ordinate must be within the range 0 to 255 (x is a cross, remember) and the y co-ordinate must fall between 0 and 191 . Should either of these parameters fall outside their ranges your program will stop with an error report.

The following program PLOTs points to the screen at random using a randomly generated colour for each PLOT. If you allow the program to RUN for a while you'll be able to see the colour resolution display taking shape. Even though computers are incredibly fast beasts you'll have to RUN this routine for about thirty minutes to witness its full potential!

```
10 REM ************************
12 REM *    GRAPHIC SCREEN    *
14 REM *    DEMO PROGRAM 2    *
16 REM ************************
20 VS 4: CLS
30 COLOUR 3,RND*15
40 PLOT RND*255,RND*191
50 GOTO 30
```

Remember, each sequence of eight horizontal pixels can only contain one PLOTting colour and each new PLOT within a given sequence will change the colour of any other pixel PLOTted in the block in question. By combining a number of PLOT statements we can use this command to draw lines and shapes. The following program demonstrates the creation of a horizontal line starting at position 100,100 and ending at position 200,100:

```
10 REM ************************
12 REM *    GRAPHIC SCREEN    *
14 REM *    DEMO PROGRAM 3    *
16 REM ************************
20 VS 4: CLS
30 COLOUR 3,15
40 FOR T=100 TO 200
50 PLOT T,100
60 NEXT T
100 GOTO 100
```

Although such a routine effectively demonstrates the action of PLOT, in real terms code like this becomes redundant when we recognise the power of the LINE command, which is faster and much easier to use. As you would expect, LINE draws a line on the graphics screen between two sets of co-ordinates. The syntax for a LINE statement is:

LINE sx,sy,ex,ey

where sx and sy are the start co-ordinates of the line, and ex and ey are its end co-ordinates. As always, the co-ordinate parameters must be within the range 0-255 for sx and ex, and 0-191 for sy and ey. The following program uses LINE to draw a box.

```
10 REM ************************
12 REM *    GRAPHIC SCREEN    *
```

```
14 REM *    DEMO PROGRAM 3    *
16 REM ***********************
20 VS 4: CLS
30 COLOUR 3,15
40 LINE 100,100,200,100
50 LINE 200,100,200,50
60 LINE 200,50,100,50
70 LINE 100,50,100,100
100 GOTO 100
```

When designing screen displays it's a good idea to sketch out the required pattern before you start to write the program. This is especially important if you want to create complicated patterns which use constant co-ordinates (as opposed to variable co-ordinates calculated by the program). By outlining your design on paper you will save yourself hours of programming 'guess-work'.

Picasso could probably have created more satisfying designs than those produced by the following program, but he would never have been able to match your micro's speed:

```
10 REM ***********************
12 REM *    GRAPHIC SCREEN    *
14 REM *    DEMO PROGRAM 3    *
16 REM ***********************
20 VS 4: CLS
30 COLOUR 3,RND*15: CLS
40 LET X1=0: LET Y1=0
50 FOR T=1 TO 50
60 LET X=RND*255: LET Y=RND*191
70 LINE X1,Y1,X,Y
80 LET X1=X: LET Y1=Y
90 NEXT T
100 PAUSE 1000
110 GOTO 20
```

The last of the MTX BASIC graphics commands that we shall look at in this section is CIRCLE. There are no prizes for guessing the function of this keyword! The syntax for the statement command is:

CIRCLE x,y,r

where x and y are the co-ordinates of the centre of the circle, and r is the radius (measured in pixels). As with all of the other graphics commands, the CIRCLE must stay within the confines of the screen, otherwise the program will stop with an error report. The following program shows the CIRCLE command in action:

```
10 REM ***********************
12 REM *    GRAPHIC SCREEN    *
14 REM *    DEMO PROGRAM 3    *
16 REM ***********************
20 VS 4: CLS
```

```
30 COLOUR 2,6
40 COLOUR 3,1: CLS
50 FOR T=10 TO 80 STEP 5
60 CIRCLE 10+T*2,95,T
70 NEXT T
100 GOTO 100
110 REM *** PRESS BRK TO EXIT PROGRAM ***
```

The trigonometric purists amongst you will probably have noticed that the circles produced by this routine are more oval than circular. Why should this be the case, since the circle's radius is constant and the computer is measuring the radius in pixels? Well, as we have already discovered, each pixel is wider than it is high, and consequently a circle created from rectangular pixels will necessarily resemble an elipse. This is undoubtedly a shortcoming of micro graphics, but one which is shared by every other home computer on the market.

By combining the three graphics commands - PLOT, LINE and CIRCLE - with the potential of PRINT and user-defined characters, you should be able to develop programming techniques which will satisfy your graphical requirements. Apart from the dramatic displays required by a games program, these commands also facilitate the creation of graphs, bar charts or pie charts which enable you to turn visually tedious textual displays into graphically satisfying representations of serious data. The MTX is particularly suited to this type of application because of the ease with which it facilitates the combination of text and graphics on the screen. Why don't you take another break and try writing some programs which exploit the flexible graphics commands we have dealt with so far?

TURTLE GRAPHICS

The graphics screen, as we have seen, is made up of 256 pixel positions across the screen and 192 positions down the screen. Lines are created by defining their start and end co-ordinates within this 256*192 grid (where position 0,0 is the bottom left-hand corner of the screen). You'll doubtlessly be intrigued to learn that this system of referencing screen positions uses 'Rectangular Cartesian Co-ordinates', which are rather similar to the grid references of maps. Although such parameters are perfectly adequate for most applications, they obviously become a trifle cumbersome in the design of more complex shapes.

A few years ago the problem of teaching young children to write computer programs was radically advanced by the introduction of a new computer language called LOGO. LOGO is easy to learn and fun to use. LOGO's accessability is primarily explained by the fact that it uses a small, friendly robot to control the creation of graphical displays. When LOGO was designed, its command structure was deliberately kept simple. Consequently the language only utilises self-explanatory commands such as TURN LEFT, MOVE, TURN RIGHT, PEN DOWN and PEN UP.

The crucial difference between this system of creating designs and the more complex Cartesian Co-ordinate system is that the movement of the LOGO robot (or turtle as it came to be known), is considered in terms of its CURRENT POSITION, and not defined in terms of a pre-defined display position. It is obviously a lot easier to tell the turtle to 'turn left and then move 10 positions forward', than trying to work out the co-ordinates of the start and finish positions of the line required.

While the graphics commands available in MTX BASIC cannot really be described as genuinely 'LOGO-like', the dialect offers four graphics statements that are based on the same co-ordinate system. This system is known as the 'Polar Co-ordinate' system because it always uses the current position as its point of reference.

The first of these 'LOGO-like' commands is ANGLE, which establishes an initial plotting direction in terms of an angle from the horizontal. Note that this command's definition makes no reference to distance, since the only function of the ANGLE statement is to specify a plot direction. The syntax of ANGLE is:

ANGLE x

Where the value of x is measured in radians. For those of you who are used to measuring angles in units of degrees (99% of the population, no doubt), we should make it clear that there are 2*PI radians in a circle (which is 360 degrees) and 1 radian is equal to 57.2957795 degrees! For a system that is supposed to simplify matters, we seem to be dealing with values that are more than a little complicated.

The secret of success when considering angles in terms of using radians is to think of 180 degrees as 1*PI radians (don't bother to work out the exact value, let the computer cope with the multiple decimal places!). Obviously any other angles can be expressed as a fraction of PI and the table below offers the degree to radian conversion for a number of familiar and useful angles.

DEGREE TO RADIAN CONVERSION

DEGREES	RADIANS
360	2*PI
270	3*PI/2
180	PI
90	PI/2
60	PI/3
45	PI/4
30	PI/6
22.5	PI/8
15	PI/12
10	PI/18
7.5	PI/24
5	PI/36
1	PI/180

An ANGLE statement with an argument of 0 determines a horizontal plotting direction facing to the right. Similarly, ANGLE PI/2 (90 degrees) establishes a vertical plotting direction facing upwards and so on. The function of ANGLE will become more apparent once we have taken a look at the next command in this sequence, the DRAW command:

DRAW x

where x is the length of the line (measured in pixel units) to be drawn. This provides us with two forms of graphical instruction, one of which specifies direction (ANGLE) whilst the other determines the distance (DRAW) to be drawn. We can now move on and draw a line utilising a routine like this:

```
20 VS 4: CLS
30 PLOT 100,100
40 ANGLE PI/2
50 DRAW 50
100 GOTO 100
```

The effect of this program is the production of a vertical line which is 50 pixels length. Although this is not intrinsically very exciting, the best is yet to come!

Although your micro will always remember the current plot direction, the PHI command can be used to alter this direction. Whenever the MTX encounters PHI it adds the value of the statement's argument to the plot direction currently stored in the computer's memory. Let's take a look at the ways in which this command's effects can be exploited. Consider how you would go about drawing a square on a piece of paper. If the square has sides which are 50 units long, you would have to complete the following procedure:

Starting with the first vertical side (ANGLE PI/2) you would draw a line whose length was 50 units (DRAW 50). Next you would turn left (PHI PI/2) and draw a second line (DRAW 50). A second turn to the left (PHI PI/2) would be followed by a third line (DRAW 50). To complete the square, a third turn (PHI PI/2) would

have to be followed by the last line (DRAW 50). The following program turns LOGO into BASIC:

```
20 VS 4: CLS
30 PLOT 100,100
40 ANGLE PI/2
50 DRAW 50
60 PHI PI/2
70 DRAW 50
80 PHI PI/2
90 DRAW 50
100 PHI PI/2
110 DRAW 50
120 GOTO 120
```

This program produces a square which is plotted without reference to screen co-ordinates (apart from the initial PLOT command in line 30). The most interesting point to note when using polar co-ordinates to alter the position of our square is that it only requires the modification of the initial PLOTting position (line 30). If you add the following lines to DEMO PROGRAM 2 you can draw 21 squares:

```
25 FOR T=60 TO 100 STEP 2
30 PLOT T,T
115 NEXT T
```

If you attempt to reproduce our last example's screen display by using LINE statements the advantages of polar co-ordinate based commands will become clear. Returning to the original version of DEMO PROGRAM 2, the orientation of the square can obviously be modified by changing the initial plotting direction (line 40). We'll leave this particular development as an exercise for our more imaginative readers!

The final LOGO-type command offered by MTX BASIC is ARC. The syntax for ARC statements is:

ARC x, theta

The ARC command enables the creation of an arc of a circle with radius x through an angle of theta (where theta is measured in radians). To draw a complete circle using ARC you would have to specify an angle of 2*PI (360 degrees). The angle (theta) established by the statement's second parameter is always added to the current plotting direction, in much the same way as a position determined by a PHI statement. This is demonstrated in the following example which adds rounded corners to our square:

```
20 VS 4: CLS
25 INK 1
30 PLOT 100,100
40 ANGLE PI/2
50 DRAW 50
60 ARC 20,PI/2
70 DRAW 50
80 ARC 20,PI/2
90 DRAW 50
100 ARC 20,PI/2
```

```
110 DRAW 50
120 ARC 20,PI/2
130 DRAW 50
140 GOTO 140
```

Once again, the time has come for you to put down your manual and revel in the fruits of your studies! We have now covered all the MTX drawing commands operative on the graphics screen. You can combine them in any way you choose (including PRINTing text on the screen), and the only limitation which can inhibit the potential of your graphical creations is your own imagination! So take a break and let rip!

ATTRIBUTING VIRTUAL SCREENS

Welcome back! There are only three MTX graphics commands which remain to be explained in this section. The first is ATTR, an exclusively graphical command and the other two are unique to MTX BASIC - DSI and CRVS.

ATTR is short for ATTRibutes and is used to alter the effect of screen-based statements like PRINT and PLOT. The syntax for ATTR is:

ATTR p,n

The statement's p parameter is used to establish the display element to be controlled, whilst the value of n actually switches control on (n=1) or off (n=0). Once an ATTRibute has been turned on, the results of all other display commands operating on the graphics screen are modified. An ATTR statement will continue to exert its influence until it is turned off again. It should be stressed the ATTR command has no effect when used on the text screen, and is only of value when utilising the graphics screen.

The effect of using p=0 is to reverse PAPER and INK colours for all subsequent PRINT statements. This process is known as reversed PRINTing. The reversed print facility is very useful when specific lines on the screen require highlighting. For instance:

```
10 REM ************************
12 REM ***      ATTRIBUTES    ***
14 REM ***  DEMO PROGRAM  1 ***
16 REM ************************
20 VS 4: CLS
30 ATTR 0,0: REM *** TURN OFF ***
40 CSR 5,4: PRINT "GAME OVER"
50 ATTR 0,1: REM  *** TURN ON  ***
60 CSR 5,6: PRINT "PLAY AGAIN?"
70 ATTR 0,0: REM *** TURN OFF AGAIN ***
80 CSR 2,20: PRINT "PRESS ANY KEY TO PLAY AGAIN "
90 PAUSE 50
100 ATTR 0,1: REM *** TURN ON AGAIN ***
110 CSR 2,20: PRINT "PRESS ANY KEY TO PLAY AGAIN "
120 PAUSE 50
130 IF INKEY$<>"" THEN  STOP  ELSE  GOTO 70
```

The flashing effect produced by this program is achieved (in lines 70 to 130), by alternatively PRINTing normal and reversed characters at the same screen position. Each time a PRINT statement is executed it wipes out any characters already displayed at that position.

The effect of using p=1 is to reverse the pattern of pixels that are switched on. This produces a different display to that created when p=0 because, in this case, the ATTR command will reverse (or turn off) only those pixels that are ON. The use of p=1 almost overPRINTS one character with another, but not quite. Overprinting implies that one character is PRINTed on top of another without wiping out the first. The ATTR 1,1 command does just this, except under circumstances in which there is already a pixel ON, and the new character also attempts to turn ON. The result of this type of graphical confation is to turn the pixel in question OFF, which often has the effect of rendering alphanumeric characters unreadable. This said, it's worth noting that this consequence of the ATTR function can be profitably exploited when applied to user-defined characters. The following example shows the effect of using ATTR 1,1 on the letters of the alphabet:

```
20 VS 4: CLS
30 FOR T=0 TO 23
40 CSR 3,T
50 FOR X=0 TO 25
60 PRINT CHR$(65+X);
70 NEXT X
80 NEXT T
90 PAUSE 1000
100 ATTR 1,1
110 FOR T=0 TO 23
120 CSR 3,T
130 FOR X=0 TO 25
140 PRINT CHR$(65+X);
150 NEXT X
160 NEXT T
170 GOTO 170
```

The last two values for the control parameter p are used with graphics-based commands like PLOT and LINE (as opposed to the character-based commands). If p=2, pixels will be unPLOTted. In other words, pixels in the INK colour will be reversed to the PAPER colour. The effect of this parameter is demonstrated by the following program:

```
20 VS 4: CLS
30 ATTR 1,0: REM TURN OFF OVERPRINT
40 FOR T=0 TO 190 STEP 5
50 LINE 0,T,255,T
60 NEXT T
70 PAUSE 1000
80 ATTR 2,1: REM TURN ON UNPLOT
90 FOR T=0 TO 190 STEP 10
100 LINE 50,T,200,T
```

```
110 NEXT T
120 ATTR 2,0: REM TURN OFF UNPLOT
130 GOTO 130
```

This program should first draw a screenful of normal horizontal lines. Lines 80-110 will then unPLOT the middle of every other line on the screen and effectively erase the lines in question.

The final legitimate value for ATTR's p parameter is p=3. When you turn this ON by coding the statement ATTR 3,1 anything plotted in the background colour takes on the foreground pigment, whilst the foreground display takes on the background colour. In other words, an unplotted point is plotted and a previously plotted point unplotted! Although you can't use this command to PRINT characters over graphics displays, you can draw graphics over previously PRINTed text.

When an ATTR 3,1 command is being processed the CLS instruction does not clear the graphics screen. Instead, these parameters allow you, for example, to change the PAPER and INK COLOUR of the display. Try the following program and you'll see how ATTR 3,1 allows you to superimpose a variety of shapes:

```
10 REM ************************
12 REM ***    ATTRIBUTES    ***
14 REM ***  DEMO PROGRAM 3  ***
16 REM ************************
20 VS 4: ATTR 3,1
25 COLOUR 0,4: COLOUR 1,15: COLOUR 2,4: COLOUR 3,1
5: CLS
30 CSR 7,9: PRINT "    THIS IS    "
40 CSR 7,10: PRINT "A DEMONSTRATION"
50 CSR 7,11: PRINT "  OF ATTR  3,1  "
60 FOR T=20 TO 170
70 LINE 30,T,215,T
80 NEXT T
90 CIRCLE 122,95,80
100 PLOT 20,30
110 ANGLE 0
120 DRAW 205
130 PHI PI/2: DRAW 130
140 PHI PI/2: DRAW 205
150 PHI PI/2: DRAW 130
160 FOR T=1 TO 15: CLS
170 COLOUR 2,RND*15
180 COLOUR 3,T
190 PAUSE 1000
200 NEXT T
210 ATTR 3,0: REM TURN OFF OVERPLOT
220 GOTO 220
```

Although all the variations of the ATTRibute statement can be combined, there are few situations in which such a fusion would be required. This said, it's worth noting that if you do decide to combine ATTR functions, the results produced by ATTR 2,1 and ATTR 3,1 are quite interesting - nothing happens! This is not as daft as it seems. As you'll see in the next chapter, your MTX offers a PLOT- SPRITE facility which makes use of the plotting positions on the graphics screen (more on this later). Secondly, when using the Polar Co-ordinate command system (ANGLE, DRAW, PHI etc.), it is possible to modify the plotting position without affecting the screen display. In other words, MTX BASIC allows the simulation of the PEN-UP and PEN- DOWN commands which are normally only available to LOGO programmers.

USING VIRTUAL SCREENS

So far we have considered the text and graphics screens as immutable fields of display. We'll now reveal that MTX BASIC makes it possible to divide the screen into a number of smaller screens, which can be defined using your micro's Virtual Screen facility.

When you first turn on your computer (and whenever you are faced with the BASIC 'Ready' prompt after RUNning a program), the MTX screen is divided into three separate (but invisible) display areas.

The top nineteen lines of the screen are referred to in MTX BASIC as Virtual Screen 1 (VS 1) - the listing screen. The next four lines are referred to as Virtual Screen 0 (VS 0) - and this is the entry screen where you enter program lines. The screen's bottom line is Virtual Screen 7 (VS 7) - the message screen, used by the computer to display error messages. Each of these screens is entirely independent, which means that no display function appearing on one screen interferes with another screen's display. A unique MTX BASIC feature facilitates the creation of customised Virtual Screens which can be defined to satisfy the display requirements of specific programs. The statement which enables the creation of user-defined virtual screens is CRVS. It takes the following format:

CRVS n,t,x,y,w,h,s

This rather formidable set of parameters determine the following display criteria:

n	Screen identification number (0-7)
t	Screen type. 0=text 1=graphics
x	X Co-ordinate of top-left corner of VS screen
y	Y Co-ordinate of top-left corner of VS screen
w	Width of virtual screen in characters
l	Depth of virtual screen in lines
s	Number of characters per full line

n ——— Let's look at each parameter in turn. The first is n, the screen number, which must be in the range 0 to 7. This said, you mustn't forget that the computer uses some of the screens represented by these values for its own purposes. To be on the safe side, you should only use screens 2, 3 and 6 when defining your own display area, as on returning to the "Ready" prompt the computer redefines VS 0, 1, 5, 7.

t ——— The second parameter, t, defines the screen type - t=0 accesses the text screen and t=1 specifies the graphics screen. Remember that you can't mix screen types. You cannot create a smaller graphics screen within the main text screen (or vice versa). Neither can you simultaneously display two different types of virtual screen. It's either text OR graphics - like oil and water, they don't mix!

x ——— The next pair of parameters (x and y), are the co-ordinates of the top left corner of your Virtual Screen. These parameters are measured in characters and lines from the top left corner of the main screen. These parameters use exactly the same principle as that employed by the CSR command to position the cursor - forty characters wide for the text screen and thirty-two characters wide on the graphics screen.

w ——— Following the x and y parameters are the values which specify the width (w) and the depth

l ——— (l) of the Virtual Screen (again measured in characters across and lines down the screen).
s ——— The final parameter determines the number of characters per full line for the screen type (i.e. 32 for a graphics screen or 40 for the text screen).

One of the most valuable features of user-defined Virtual Screens is that they operate independently from the main screen and each other. For example, when using the graphics commands, plotting co-ordinates start at 0,0 in the bottom left corner of every virtual screen that has been defined, NOT the bottom left corner of the main screen. Similarly, co-ordinates for CSR statements take their start positon from the top corner of the current Virtual Screen.

When using graphics screens, the colour of each Virtual Screen can be independently modified, using straightforward COLOUR commands. On the text screen, however, you are still restricted to a single PAPER and INK colour for the entire screen - in general the use of virtual text screens is more restricted than equivalent graphics screens.

Once a Virtual Screen has been defined (using CRVS), it can be called into use within a program by using a VS (screen number) statement. All subsequent commands will be directed to the specified screen until another VS instruction is encountered.

To give you a taste of the potential offered by the Virtual Screen facility, the fcllowing example creates a television screen on your television screen! It's best if you do a cold start before typing in this program, so RESET the machine and away you go:

```
10 REM ************************
12 REM ***  VIRTUAL SCREEN  ***
14 REM ***   DEMO PROGRAM 1 ***
16 REM ************************
20 VS 4
30 COLOUR 0,7: COLOUR 1,5: COLOUR 4,5
40 COLOUR 2,7: COLOUR 3,5: CLS
50 PLOT 50,30
60 ANGLE 0: DRAW 150
70 PHI PI/2: DRAW 130
80 PHI PI/2: DRAW 150
90 PHI PI/2: DRAW 130
100 PLOT 70,47: PHI PI/2: DRAW 82
110 PHI PI/2: DRAW 90
120 PHI PI/2: DRAW 82
130 PHI PI/2: DRAW 90
140 CIRCLE 175,140,10
150 CIRCLE 175,110,5
160 CIRCLE 175,80,5
170 CIRCLE 175,50,5
180 CRVS 2,1,9,7,10,11,32
190 COLOUR 0,7: COLOUR 1,5: COLOUR 4,5
200 COLOUR 2,7: COLOUR 3,5
```

```
210 VS 2
220 PRINT CHR$(27);"P";: REM  TURN ON AUTO SCROLL
230 FOR W=1 TO 10: FOR T=0 TO 25
240 PRINT CHR$(65+T);
250 NEXT T: NEXT W
260 PAUSE 1000
270 VS 4: CSR 4,2: PRINT "BACK TO THE MAIN SCREEN"
280 PAUSE 1000
290 VS 2: CLS : CSR 0,2: PRINT "      AN     EXAMPLE
   OF         VIRTUAL    SCREENS"
300 GOTO 300
```

Obviously this crude demonstration program could be enhanced by the addition of text PRINTed to the 'screen within a screen'. You could even move up-market and code your own graphics to this Virtual Screen, although remember that your plotting must be contained within the confines of the smaller

Virtual Screens in the textual mode can be usefully developed in conjunction with DSI, the final command to be discussed in this chapter.

DSI

DSI (short for Direct Screen Input), is another command unique to MTX BASIC. As its full name implies, DSI allows us to enter any type of data from the keyboard. The command is radically different to other forms of data entry (INPUT, EDITOR and INKEY$), since it permits ANYTHING typed from the keyboard to be acted upon and/or displayed to the screen. This includes cursor controls, colour changes and any form of input. The DSI facility is terminated by pressing the <RET> key. The following program allows you to roam freely around the screen. When you have finished, press <RET> and the program will end. This is your chance to let rip at the keyboard without worrying about the dreaded error messages!

```
10 REM ****************************
12 REM *** DIRECT SCREEN INPUT ***
14 REM ***   DEMO  PROGRAM  1   ***
16 REM ****************************

20 VS 5: CLS
30 DSI
40 STOP
```

DSI offers access to the ESC and CTL sequences listed below in addition to the display facilities provided by the standard character set. However, if you decide to utilise the ESC/CTL codes you must ensure that you enter the codes in the correct sequence, otherwise the MTX will return an SE.B error. When using CTL sequences you should simultaneously press the CTL key and the nominated key. When using ESC sequences you should first press the ESC key, followed by the nominated key.

EFFECTIVE SEQUENCES DURING DSI

SEQUENCE	EFFECT
CTL W	Tab back one position
CTL]	Set page mode
CTL \	Set scroll mode
CTL ^	Turn cursor on

CTL _	Turn cursor off
CTL D	Change PAPER colour. Letters A to O then selects colours 1 to 15
CTL F	Change INK colour. Letters A to O then selects colours 1 to 15

ESCAPE SEQUENCES

ESC I	Insert a line at the current cursor position
ESC J	Delete a line at the current cursor position.
ESC K	Duplicate the current line on the line below. This deletes the current contents of the new line

See also Appendix 2

Whilst DSI is a unique and flexible enhancement of BASIC, the statement does not actually allow you to enter information into your program. To do this you must make use of the SPK$ command which reads this type of data into a string variable which is stored for further use. Our next example uses DSI to enter data on to a Virtual Screen and SPK$ to read the screen into a variable:

```
5 REM  ******************
10 REM *** DSI DEMO 2 ***
15 REM ******************
20 VS 5: CLS : DIM A$(240)
30 CRVS 2,0-0,20,0,20,12,40
40 INK 3: PAPER 1
50 VS 2: CLS
60 DSI
70 CSR 0,0
80 FOR T=1 TO 240
90 LET A$=A$+SPK$
100 NEXT T
110 CLS : VS 5
120 CSR 0,0: PRINT "YOU TYPED :-";A$
140 GOTO 140
```

When you RUN this program, you can display anything you want on the Virtual Screen we have created on the right-hand side of the full screen. Your display always starts at the top left corner of VS 2, but the program enables you to relocate it at any position required. Once you are happy with your input you should press <RET> to end the editing session. The program will pause for a second and then store the entire virtual screen; this is accomplished by means of the SPK$ function and a FOR...NEXT loop. Finally, the program rePRINTs your data on the full screen.

When creating your own programs you could, of course, use this type of code for other purposes. The implementation of this form of data entry is clearly more flexible than the use of INPUT or EDITOR statements because it allows you to format your entry in any way you please. As you know, INPUT statements will only allow entry on a single line and do not permit the use of a comma within a single entry. There are no such problems with DSI!

CHAPTER 10 : SPRITE GRAPHICS

SPRITES AND HOW TO CONTROL THEM

Your MTX is capable of displaying up to thirty-two sprites on the screen at any one time. The sprites you define can be one of three sizes and each sprite can be allocated to any of the sixteen MTX colours. Sprites can move independently of a BASIC program or controlled from within a program. Three dimensional effects can be simulated with the use of the built-in sprite priority which enables sprites to be moved in front of one another. Sprites can even be controlled when they are not visible on the screen!

In the pages that follow we shall examine the commands which facilitate the creation and animation of sprites. These features of MTX BASIC are both powerful and sophisticated, so don't be discouraged by their apparent complexity. The fact of the matter is that it's relatively simple to create fast action games in BASIC once you've developed a working knowledge of the mechanics of sprite graphics. The essential point to remember is that because all sprite commands are interrelated, it's not possible to activate a sprite until the entire sequence of relevant commands have been grasped.

CTLSPR (parameter),(value)

The CTLSPR command (ConTrol SPRite) assigns global values which determine the nature of all subsequent sprite creations. Any value set by this command affects all of the sprites in the same way. This contrasts with the SPRITE command (see below) which will only affect a specified sprite. Only one 'parameter' can be specified within each CTLSPR command, and it must be a value 0-6. The consequences of each 'parameter' value are itemised below, along with the appropriate ranges required by the statement's second parameter (value):

parameter= 0
value range: 0-255

controls the speed of movement of all the sprites, where 1 is the fastest speed and increasing numbers produce lower speeds up to 255. However, the slowest speed is obtained by using a value of 0! This parameter is used in conjunction with the SPRITE command to determine the overall speed of each sprite. CTLSPR parameter=0 will determine how often the sprites are moved, a value of 1 will give a movement every master interrupt cycle, a value of 2 will give a movement every other cycle, and so on. As you will see later, the SPRITE command's distance parameter determines the size of each movement. Remember that whilst the CTLSPR command establishes a general speed factor, the SPRITE command determines a different speed for each sprite.

parameter=1
value range: 0-7

the CTLSPR command is used to control the range of movement by which all sprites are constrained. A value of 0 will result in no movement, whereas a value of 7 will produce a movement of 7 pixels in the appropriate direction. The value of this parameter is that it can be used in conjunction with the MVSPR command to move any sprites a standardized distance in any direction. Parameter 1 has no effect when used with the SPRITE command, only when it is used with MVSPR.

parameter=2
value range: 1-32

tells the computer how many sprites are to be shown on the screen. This value must be the total number of sprites that you want to use and must be declared before you use any sprites. Failure to do so will cause the program to stop with an error message. If, for instance, you try to use four sprites after declaring only three then the program will stop when the first attempt is made to display the fourth sprite.

So, as you design a program you must keep a close count of the number of sprites on the screen. Although you have the option of declaring all 32 sprites whether you require them or not, such a decision will considerably slow down your program. The computer will act as though it controls all 32 sprites, even though only 3 or 4 may be actually in use. The MTX will also allocate memory, for every sprite declared in much the same way as it allocates memory when dimensioning an array. So unnecessary declarations of sprites should be avoided by establishing your programming requirements at the planning stage.

parameter=3
value range: 0-32

declares the number of orbiting sprites required in a program. The range of values for this parameter is 0 to 32, although the value allocated must not be greater than the total number of active sprites.

This is as good a time as any to outline the characteristics of an orbiting sprite. If so instructed a sprite will orbit, and if it goes off the edge of the screen, will shortly (depending upon its speed) reappear on the opposite side of the screen, as if it had gone all the way round the back of the TV set!

parameter=4
value range: 1-32

the CTLSPR command allows you to define any one sprite as a so-called PLOT SPRITE. A plot sprite behaves differently from the other sprites. Instead of directing its movement via the normal sprite commands (SPRITE, MVSPR and so on) plot sprites follow the points and lines created by other MTX BASIC graphics commands (i.e. LINE, DRAW, PLOT, CIRCLE etc.) The centre of such a sprite will always be determined by the position of the currently plotted pixel. Only one plot sprite can be displayed on the screen at any given time, although it is possible to switch alternatively between a number of different sprites using the MVSPR command.

parameter=5
value range: 0-32

the CTLSPR command defines the total number of independently mobile sprites. The actual movement is defined by the SPRITE and ADJSPR commands. Whilst it is possible to declare more self motivated sprites than you intend using at any one time, you cannot assign a value in excess of the total number of activated sprites (defined by parameter=2). Your program will stop with an error message if you attempt to create more moving sprites than the number of active sprites previously declared.

parameter=6
value range: 0-3

there are four variations of sprite shape available to MTX programmers. Assigning a value of 0 to this parameter will display the standard eight by eight pixel sprite. A value of 1 will double the size of a standard sprite whilst a value of 3 will produce a double sized sixteen by sixteen sprite. All sprites must be the same declared size, and it is not possible to mix different sized sprites on the screen.

GENPAT

The GENPAT command is the MTX's GENeration PATtern creation command. As we have seen, it can be used in the definition of user-defined characters, but also enables programmers to control the shape of a sprite. Compared to the contortions that some computers go through when defining a character or sprite, GENPAT is a delight to use. The syntax for the command is as follows:

GENPAT p,n,d1,d2,d3,d4,d5,d6,d7,d8

Although at first glance the statement's parameters appear complicated, establishing parameters for GENPAT statements is really quite simple. The parameter p specifies the type of object to be defined.

p=0 Redefines one of the ASCII characters, where n takes the value of an ASCII code in the range 32 to 127.

p=1 Redefines one of the non-ASCII characters, where n takes a value between 129 and 154. These 25 characters can be PRINTed to the screen using the CHR$ function. They can be used when a program requires the complete standard character set to be intact. The first fifteen can be summoned by Function keys 2 – 16.

p=2 When parameter p is equal to 2 you can create up to eight multi- coloured characters, where n takes the value (in the range 147- 154) of one of these characters. Having previously defined the shape of one of these characters (using p=1), you can define the INK and PAPER colours for individual rows of a character. When using p=2, the parameters d1 to d8 are used to define the colours for a single horizontal row, which is established by means of the MTX colour codes. The total for each row is:

VALUE=16 * PAPER + INK

The creation and use of user-defined characters is discussed in chapter 8.

p=3 Defines an eight by eight pixel sprite shape. This is the same size as the normal user-defined characters displayed on the graphics screen. You can define up to 128 shapes of this size, although only a maximum of 32 sprites can be displayed at any one time. You can, however, create sophisticated animation routines by using the ADJSPR command to mutate a single sprite through several shapes.

p=4 As well as the standard eight by eight pixel sprite you have the alternative of using sprites four times that size (sixteen by sixteen pixels). Such sprites can be created by a separate definition of the four eight by eight blocks. The parameter p=4 is used to define the top left-hand corner of your creation.

p=5 Defines the bottom-left block of the sixteen by sixteen sprite.

p=6 Defines the top-right block

p=7 Defines the bottom right block

When defining the larger sprite you can only create a total of 32 shapes (128/4=32).

Moving on to the second value of the GENPAT's argument, the n parameter takes a slightly different role contingent upon the value of p, but always serves as a pointer for the shape. For example, when defining an ASCII character, the value of n is determined by the ASCII code of the character to be re-defined. Similarly, when defining the shape of a sprite, n becomes the shape pointer (or number) used by SPRITE, MVSPR and ADJSPR statements. The table below lists the ranges of n in relation to the values of p:

p	n
0	32 - 127
1	129 - 154
2	147 - 154
3	0 - 127
4	0 - 31
5	0 - 31
6	0 - 31
7	0 - 31

The statement's final parameters (d1 to d8), contain the data of the shape to be created. The creation of characters has already been explained at some length in chapter 8, so we'll only provide a brief resume in this chapter. If you have any problems related to the concept of user-defined characters refer to the

Each value assigned to the d1-d8 parameters describes a horizontal row of eight pixels, where each binary digit (or bit), represents one pixel in the row. By totalling the bits which are 'set' (or 'on') in each row, we can obtain the appropriate value for each of the eight remaining parameters. The parameter d1 determines the value of the top row of the character and d8 the value of the bottom row. The process of establishing values for these parameters is the same for characters, eight by eight sprites and each quarter of the sixteen by sixteen sprites.

The only situation that requires an alternative process to determine the values for these parameters is when GENPAT's first parameter is equal to 2. This process is explained at some length in chapter 8, but to refresh your memory remember that each row of characters is calculated using the formula:

value of parameters d1 to d8 = 16 * PAPER + INK

Of necessity, MTX sprite statements are interdependent. Consequently it's virtually impossible to demonstrate any of the commands we've discussed so far without precipitating more confusion than clarification. This said, we can still offer an example of GENPAT format which will ultimately clarify the effects of the command! The statement below is the syntax required to define a sprite as a box:

GENPAT 3,0,255,129,129,129,129,129,129,255

This statement will create a shape whose pointer is 0. At this stage, such information will not mean very much to you since nothing can actually be seen on the screen. But never fear! Once we've tackled one further command we'll have enough information to actually get a sprite up and moving on the screen, so persevere a little longer!

SPRITE

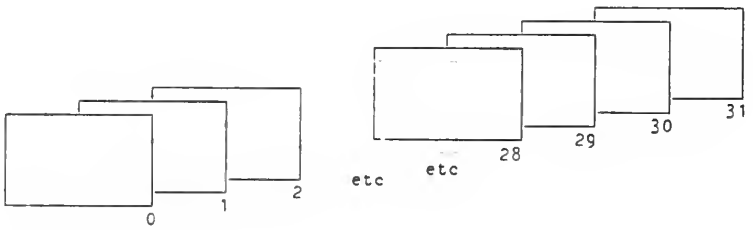
The sprite command is, in many ways, the most important command of this series. Altogether, SPRITE has seven parameters which between them define the shape to be used, its position on the screen, and the speed and colour of the sprite. The syntax for the SPRITE command is as follows:

SPRITE n,pat,xp,yp,xs,ys,c

Parameter n — This parameter is used to tell the MTX which sprite is to be affected by the SPRITE command, this value assigned to n is the pointer number of the desired sprite which must be within the range 1 to 32.

Parameter pat — This parameter is used to determine the shape (or pattern) to be allocated to sprite n. The pattern must have been previously created by the GENPAT command and with standard eight by eight sprites, p can lie within the range 0 to 127. A maximum of 32 larger sprite patterns can be defined so the pat parameter will have to be within the range 0 to 31 when using sixteen by sixteen sprites. Remember, you cannot display different sized sprites on the screen at the same time.

Params xp,yp — These parameters are used to position the sprite on the sprite plane. You can think of each sprite as having its own plane upon which to move.



In this way it's possible to use sprite graphics to create three dimensional effects, but one miracle at a time! For now we'll examine the more conventional exploitation of the sprite planes.

Each sprite plane comprises a grid of 8192 by 8192 pixels, and a specified sprite can be placed anywhere on this grid. The co-ordinates range from -4095 to +4095 in both the horizontal and vertical directions, where the co-ordinate 0,0 is normally the bottom left corner of the graphics screen. However, it should be noted that since the screen is only 256 pixels wide by 192 pixels high it is not possible to display the entire sprite plane at any given momemt. Consequently it's best to think of your TV screen as a 'window' which enables you to see a section of the sprite plane.

If the parameters xp and yp are assigned values that cause the sprite to be placed outside the visible screen range, you'll obviously not be able to see the sprite in question. On the other hand, this aspect of the sprite

plane can be used to good effect since it enables you to smoothly scroll the sprite on to the screen from a hidden position.

So, the range of values that can be assigned to xp and yp is -4095 to +4095. However, it is advisable to note that if you assign a value to xp that is outside the range of 0 to 255, or if yp is assigned a value outside the range 0 to 191, it is quite likely that the sprite will not actually be displayed on the screen. Later in this chapter we'll show you how to move the 'window' around the sprite plane area by using the VIEW command.

Params xs,ys The parameters xs and ys are used to determine the speed of independent movement in the x and y directions. Both of these parameters use values within the range -128 to +127. If xs=0 or ys=0 there will be no movement in the specified direction, so you will need to assign 0 to these parameters if you don't want the sprites to move independently.

Although there are three more sprite commands yet to be detailed, we have (at last!) reached a point where we can place a sprite on the screen. Let's make our spritely move!

PLACING A SPRITE ON THE SCREEN

1. Use the command CTLSPR to give the computer general details about the sprite(s) you wish to use; the number of sprites to be displayed, their size etc.
2. Use the GENPAT command to define the shape of the sprite(s).
3. Use the SPRITE command to display the sprite(s) on the screen at the desired co-ordinates.

The following program places a single sprite on to the middle of the screen:

```
10 REM ****************************
20 REM *** SPRITE DEMONSTRATION ***
30 REM ***      PROGRAM  1        ***
40 REM ****************************
50 VS 4: CLS
60 CTLSPR 2,1
70 CTLSPR 6,0
80 REM *** SHAPE NO. 1 SPACE INVADER ***
90 GENPAT 3,1,20,28,62,127,62,28,42,73
100 SPRITE 1,1,127,96,0,0,1
110 GOTO 110
115 REM *** PRESS BRK  TO EXIT PROGRAM ***
```

When you RUN the above program you'll be confronted by a black Space Invader positioned in the middle of the screen. Line 50 accesses the graphics screen and then clears it. The next two lines (60 and 70) set the CTLSPR parameters we wish to change and line 60 tells the MTX that only one sprite is going to be used. Line 70 makes it clear that this sprite will be of the standard eight by eight pixel variety. Any parameters that aren't altered will be set to zero by default. Line 90 defines sprite shape 1 to be the invader and the following line (100) uses the SPRITE command to set sprite 1 to shape 1.

So our first sprite is positioned in the middle of the screen and, as no speed values have been defined, our creation is stationary. The final parameter in this SPRITE command sets the colour of the sprite and as the value of one has been used the sprite is black. Line 110 is used to keep the program running, otherwise the computer would immediately return to the text screen without giving us a chance to see the sprite! Thus, you'll need to press the BRK key to exit the program.

Now that there's a real live sprite on the screen you could try adding some lines and experiment with your new found knowlege. For example, the sprite can be made double sized by changing line 70 to:

```
70 CTLSPR 6,1
```

and you can move the sprite by adding the following lines:

```
95 FOR XP=0 TO 255
100 SPRITE 1,1,XP,96,0,0,1
105 NEXT XP
110 GOTO 95
```

Similarly, the sprite will change colour if you make the following lines:

```
95 FOR T=0 TO 15
100 SPRITE 1,1,127,96,0,0,T
105 PAUSE 1000
110 NEXT T
115 REM *** PRESS BRK  TO EXIT PROGRAM ***
120 GOTO 95
```

Adding these lines will display the sprite in all of the available colours. You can also change the shape of the sprite by changing line 90. Here are a few suggestions. Type them in (one at a time of course), and note the results.

```
90 GENPAT 3,1,0,3,14,31,126,31,14,3
```

or

```
90 GENPAT 3,1,60,126,219,255,231,126,36,60
```

or

```
90 GENPAT 3,1,0,0,246,33,126,120,112,0
```

or

```
90 GENPAT 3,1,16,186,124,16,56,124,56,16
```

or

```
90 GENPAT 3,1,0,68,124,124,124,84,16,16
```

It's possible to display up to thirty-two sprites on the screen at any one time. The following program verifies this claim but only uses eight different shapes:

```
10 REM ********************************
20 REM ***  SPRITE DEMONSTRATION  ***
30 REM ***      PROGRAM  2        ***
40 REM ********************************
50 VS 4: CLS
60 CTLSPR 2,32
70 CTLSPR 6,0
80 GENPAT 3,0,20,28,62,127,62,28,42,73
```

```
90 GENPAT 3,1,0,3,14,31,126,31,14,3
92 GENPAT 3,2,60,126,219,255,231,126,36,60
94 GENPAT 3,3,0,0,246,33,126,120,112,0
96 GENPAT 3,4,16,186,124,16,56,124,56,16
98 GENPAT 3,5,0,68,124,124,124,84,16,16
100 GENPAT 3,6,165,66,66,36,24,60,90,129
102 GENPAT 3,7,16,16,84,124,124,124,68,0
110 FOR SP=32 TO 1 STEP -1
120 SPRITE SP,RND*8,40+SP*5,20+SP*5,0,0,RND*16
130 PAUSE 1000
140 NEXT SP
150 GOTO 150
160 REM *** PRESS BRK TO EXIT PROGRAM ***
```

The shape and the colour of each sprite is chosen at RaNDom in line 120, which is itself within a FOR...NEXT loop that counts through all thirty-two sprites. Notice that as each sprite is displayed it appears to be in front of the previous sprites on the screen, thus achieving a three dimensional effect. There are two other aspects of this program which are worthy of note. Firstly, some of our random sprite colours clash with the background colour of blue. In your own programs you should choose the colours carefully and not at RaNDom as has been done here. Secondly, you can use calculated parameters within any of the sprite-related commands, you're not restricted to literally quoted values.

Let's now move on to the last of the three sprite commands:

MVSPR p,n,d

The MVSPR statement is a general purpose command that can simultaneously perform a variety of functions, depending on its parameters. Four different functions are available and they are detailed below:

parameter	action
1	move sprite
2	select sprite pattern
4	redirect sprite movement
8	plot a point at the centre of the sprite

The second value, n, is the number of the sprite that you are dealing with. The third value, d, contains the value of the action that is specified by p, and will have different ranges depending upon the required action:-

p	range of d
1	0-8 where 0 and 8 are the same direction
2	0-127 (8*8 sprites) 0-31 (16*16 sprites)
4	0-8 where 0 and 8 are the same direction
8	not relevent

When using the plot-sprite parameter, 8, the value of d will not have any direct effect. Simply specifying p=8 is sufficient to cause a pixel to be plotted at the current position of the centre of the specified sprite.

One problem that may confound you when making use of MVSPR centres around establishing a value for d, which must be specified with these multiple instructions. If, for example, you wish to move in direction 3 whilst at the same time changing to sprite shape 3 then you might conceivably attempt to add the two numbers together to give a value for d of 6. The complete instruction would then be:

MVSPR 3,1,6

However, such a statement will undoubtedly cause the MTX a great deal of confusion. Being a mere computer, it is unable to translate the value 6 into 3+3. This is, when you think about it, quite reasonable since 2+4=6 and 5+1=6 and 1+5=6 etc,etc. In other words, the computer is unable to decipher the value 6 into your required combination of responses. This said, it IS possible to combine activities within a single command, but only certain combinations are permissable. In particular, any ONE of the first three parameters (i.e. p= 1,2 or 4), can be combined with the final parameter (p=8), but no other combination is permissable. In particular, combinations of the first three parameters are not permissable because of the inherent confusion over the resulting combination of n values.

ADJSPR p,n,v

The ADJSPR command allows you to ADJust any one of the parameters of the SPRITE command. Because it is only concerned with a single parameter this command will be executed considerably faster than the full SPRITE command. When using ADJSPR, the parameter p takes the following values:

p	meaning
0	sprite pattern
1	sprite colour
2	sprite X position
3	sprite Y position
4	sprite X speed
5	sprite Y speed

The second value used by this statement is n, which is simply the number of the sprite to be modified.

The final value, v, is the figure by which you will ADJust the appropriate parameter. The range of values that can be used for v will, of course, depend upon which of the parameters is to be altered. The following list details ranges for all of the parameters:

p	v
0	0-31 16*16 pixel sprite 0-127 8*8 pixel sprite
1	0-15
2	0-255
3	0-255
4	0-255 128-255 indicates movement to the left
5	0-255 128-255 indicates movement downwards

```
10 REM ********************************
20 REM ***  SPRITE DEMONSTRATION  ***
30 REM ********************************
50 VS 4: CLS
60 CTLSPR 2,1
70 CTLSPR 6,0
80 REM ***  SHAPE NO. 1 SPACE INVADER ***
90 GENPAT 3,1,20,28,62,127,62,28,42,73
100 SPRITE 1,1,127,96,0,0,1
110 GOTO 110
```

When you have typed in the above program and RUN it you should see the all too familiar black Space Invader in the middle of the screen. Adding the following lines will allow the sprite to move across the screen from left to right:

```
110 FOR T=0 TO 255
120 ADJSPR 2,1,T
130 NEXT T
140 GOTO 110
```

The program continues to repeat lines 110 to 140 until you press the BRK key. Try changing line 120 to:

```
120 ADJSPR 3,1,T
```

When you RUN the modified program you will see the invader move from the bottom of the screen to the top. Unlike the previous example, this program doesn't repeat but stops with an error report! The reason for the error lies in line 110, which should be changed so that the value of T always falls within the correct range of 0-219.

```
110 FOR T=0 TO 219
```

Now your program should RUN quite happily until you press the BRK key. You may be wondering why the range of positions with the SPRITE command is -4095 to +4095, since the ADJSPR command can only use 0-255 horizontally and 0-219 vertically. As we have already seen, the SPRITE command is used to position a sprite anywhere upon the sprite plane. However, the ADJSPR command can only be used to make adjustments to sprites that are visible on the screen. So, the value used to position a sprite using ADJSPR are the screen co-ordinates, not the sprite plane co-ordinates.

Under these circumstances it is reasonable to expect an error message if you try to move a sprite that is not visible. However, you should note that you cannot make ANY ADJustment to a non-visible sprite. Attempting to do so will always stop your program with an error report, which is definitely unhelpful if you're trying to defend the civilised world from hoards of alien invaders! To satisfy yourself that this is true, first change line 100 to:

```
100 SPRITE 1,1,128,-96,0,0,1
```

Now, if you change the parameter value in line 120 to, for example, 5 you will receive an error report as soon as you RUN the program. This is because the sprite is located at position 128,-96 which is off the bottom of the screen.

We've chosen to discuss parameter problems at this point in an attempt to highlight the interrelated nature of the MTX's sprite commands. As we've progressed through this chapter you've undoubtedly realised that we've used variations of the demonstration program. This approach, although frowned upon by the advocates of structured programming, is acceptable when exploring your computer's capacity. This said, it should be admitted that, to a certain extent, you've been lead down the garden path in order to illustrate one of the biggest problems that plagues program constructon. When you change one part of a program you MUST ALWAYS determine the effect of the change on the rest of the program. A single change to a program often causes unpredictable effects elsewhere and this type of bug will often be the hardest to locate and correct.

The deliberate mistake in the last example is simply that we forgot to tell the computer that we wanted our sprite to orbit. When the sprite goes off the screen it is 'forgotten' about and then generates an error report the next time it has to perform an ADJSPR!! Restore line 120 and add the following line:

```
65 CTLSPR 3,1
```

The complete demonstration program should now look like this:

```
10 REM*****************************
20 REM*** SPRITE DEMONSTRATION ***
30 REM***      PROGRAM          ***
40 REM*****************************
50 VS 4: CLS
60 CTLSPR 3,1
65 CTLSPR 3,1
70 CTLSPR 6,0
80 REM*** SHAPE NO. 1 SPACE INVADER ***
90 GENPAT 3,1,20,28,62,127,62,28,42,73
100 SPRITE 1,1,128,-96,0,0,1
110 FOR T=0 TO 219
120 ADJSPR 3,1,T
130 NEXT T
140 GOTO 110
```

Again, try changing the parameter for the ADJSPR command and you will now see that invisible sprites can be altered. In fact, it is now possible to make any adjustments to the invisible sprite. When you have several sprites within a program, some orbiting and others that don't, then you will have to keep track of the non-orbiting sprites before using the ADJSPR command.

VIEW direction,distance

The VIEW command allows you to move the graphics screen around the sprite plane. The distinction between this command and sprite movement commands like ADJSPR, is visually subtle but, none the less, important.

The sprite commands SPRITE, ADJSPR, MVSPR and so on allow you to control the position of a single sprite anywhere within the sprite plane. Similarly, the VIEW command allows you to move the graphics screen around on the sprite plane, thus allowing off-screen sprites to become visible (assuming you tell the MTX that the sprites are orbiting).

When using the VIEW command sprites remain in the same position on the sprite plane and the VIEWing area is moved in the chosen direction. VIEW can also be used very effectively when manipulating a complicated multiple-sprite whose movements would normally appear jerky and unrealistic. The solution to this problem is to move the background, leaving the sprite at the same sprite plane co-ordinates, thus giving the impression of movement.

The same rules apply to non-orbiting sprites when using VIEW. Non-orbiting sprites that go off the screen cannot then be altered with ADJSPR. Equally, non-orbiting sprites that disappear from the screen when using VIEW will not reappear when the screen is returned to its former position – the sprite will be lost forever. Of course, this is not the case with sprites that have previously been declared as orbiting sprites, as you can see from the following program (our friendly space-invader again!):

```
10 REM *********************************
20 REM ***  SPRITE DEMONSTRATION  ***
30 REM ***      PROGRAM 4          ***
40 REM *********************************
50 VS 4: CLS
```

```
60 CTLSPR 2,1
65 CTLSPR 3,1
70 CTLSPR 6,0
80 REM *** SHAPE NO.1 SPACE INVADER ***
90 GENPAT 3,1,20,28,62,127,62,28,42,73
100 SPRITE 1,1,127,96,0,0,1
110 IF INKEY$="M" THEN  VIEW 0,10
120 IF INKEY$="N" THEN  VIEW 4,10
130 IF INKEY$="A" THEN  VIEW 6,10
140 IF INKEY$="Z" THEN  VIEW 2,10
150 GOTO 110
```

When you RUN this program you'll see the invader in the centre of the screen. Pressing the 'M' key will move the graphics screen to the right, making the sprite appear to move left. Similarly, the 'N' key will move the screen to the left, 'A' moves the screen upwards and 'Z' moves it down. In all cases the sprite will appear to move in the opposite direction!

Using the VIEW command you could create a marvelous STAR-TREK game, searching through the galaxies for new alien life forms, boldly going where no computer has gone before! VIEW is another command unique to MTX BASIC and shows, once again, how powerful and sophisticated the BASIC language can be if a little thought is put into the design of the computer.

PLOT SPRITE

We have mentioned the PLOT SPRITE a number of times in passing through the past three chapters. Now is the time to examine this phenomenon in greater detail.

Any individual sprite can be defined as a PLOT SPRITE. Once defined, the sprite in question will follow any plotting to the graphics screen, and the effects can be very interesting. For example, having defined a sprite as a plot-sprite you can draw a circle and the sprite will follow the circle as it is drawn. The next program uses the command to display a disoriented spider weaving its web:

```
10 REM ************************
15 REM ***    PLOT SPRITE    ***
20 REM ***  DEMO PROGRAM 1   ***
25 REM ************************
30 VS 4: CLS
40 CTLSPR 2,1: CTLSPR 4,1
50 GENPAT 3,1,20,28,62,127,62,28,42,73
60 SPRITE 1,1,127,96,0,0,1
70 LET X=RND*255: LET Y=RND*191
80 LET X1=RND*255: LET Y1=RND*191
90 LINE X,Y,X1,Y1
100 LET X=X1: LET Y=Y1
110 GOTO 80
```

All the work of drawing the RaNDom web is performed by lines 80 to 110, which is about as compact a

routine as you'll find. To carry out the same functions on any other computer would involve pages of program and the end result would be far too slow to be realistic. On your MTX all the hard work is done for you by the computer, which is just how it should be in the nineteen eighties!

When we discussed the ATTR command in the previous chapter we mentioned that nothing happens to the result of plotting or drawing when combining ATTR 2,1 and ATTR 3,1. We then went on to say that this wasn't at all crazy but actually very useful when using plot-sprites. Now we'll see how, and why. Insert the following line to our plot-sprite program and you will see that our plot-sprite/spider, by all appearances, meanders around the screen with no visible means of support:

```
45 ATTR 2,1: ATTR 3,1
```

In order to return the program to its original state, with the sprite drawing the LINEs, it is not sufficient to simply delete line 45. Remember that ATTRibutes will remain 'on' or 'set' until such time as they are switched off (i.e. until such time as the value of the second parameter of the ATTR command is equal to zero). Consequently you will need to change line 45 to read:

```
45 ATTR 2,0: ATTR 3,0
```

In this introductory manual it is not possible to examine all of the complexities of high-level graphical generation. However, let's hope that this graphics section has started you along the road of understanding. The full graphics capabilities of the MTX now await your every command! Whether you use your computer for business or pleasure, for VAT returns or Space Invaders, the graphics facilities should never let you down, provided that you have a clear understanding of the task that you wish to fulfil. First create the picture in your mind and then create it on the screen!

The last program in this chapter is a simple game using sprites for all of the moving objects. You may well like to alter and improve upon it, perhaps by adding some scenery in the background, or even interesting sound effects. Don't forget to do a cold start before you start keying-in. Remember to SAVE the finished program onto a cassette before you RUN it, just in case it is more than the spaceships that crash!

The instructions are simple. Use the keys 4-LEFT,8-RIGHT and 6-FIRE and shoot down as many invaders as possible. If more than 10 get past your defences then the game, and the world, is lost.

CHAPTER 11 : THE NODDY LANGUAGE

NODDY

Your MTX has three languages resident in its memory - Assembler, Noddy and BASIC. In this chapter we will be taking a look at the Noddy language. This is a language that has been designed to simplify text handling and, as it only uses eleven commands, it is very easy to get to grips with. In order to exploit this language to the full you must get used to the idea of coding it in conjunction with BASIC. In essence, Noddy provides you with a simple way of storing and displaying textual information. In other words you can create well designed textual screens and wonderful adventure games within a BASIC program by calling on a specific Noddy file. It is very simple to switch from BASIC to Noddy without losing any files resident in memory.

To access the Noddy language type in as a direct command NODDY (or NODD.) and press the <RET> key. The screen will display:

Noddy>

This 'Noddy>' prompt is the equivalent of the BASIC Ready prompt. It is there to tell you that the MTX is ready and awaiting instructions. The first thing that the Noddy language expects us to do is to create a file name, which is simply a name or label given to each specific Noddy page of text. Let's call our first file EG, so type in EG and press <RET>. If you typed this in lower-case (as opposed to capitals), you will have to use the lower-case version of a filename each time you want to recall that particular file, since the Noddy language distinguishes between the two.

Having created our file the screen is cleared and the filename displayed at the top left-hand corner of the screen. Fill the screen with information, but do not press either the CLS or RETURN keys. You can move around the screen using the cursor control keys, the BS key and line feed key. If the CLS key is pressed (accidentally) you will lose the file that you are currently working on.

When you have got used to the 'feel' of using the Noddy screen and want to leave the file press the <RET> key. The difference between the effect of using the <RET> key and the CLS/ENT key is that whilst the CLS/ ENT key erases the file from memory, the <RET> key simply exits the file and stores it in the MTX's memory (thus making it possible to come back to it at a later date).

Once the <RET> key has been pressed you will leave the file and the 'Noddy>' prompt and cursor will reappear at the bottom of the screen. We now have a file called EG which can be accessed at anytime. However, the file will be lost if you:

a) switch off the machine
b) NEW the memory (when you are in BASIC)
c) RESET the machine (using the keys placed either side of the space bar)

To check the file is in memory you can use the DIRectory command, DIR which lists all the current Noddy files. This command must be entered in capitals, otherwise the MTX will think you are creating a new file. So, type in DIR and press <RET>. The screen should clear and the filename EG be displayed in the top left hand corner of the screen. To return to the EG file all you have to do is type in EG and press <RET>.

Before you read any further try opening a few more files and then displaying their names using DIR, once you feel perfectly at home with this process clear the MTX's memory (using one of the methods mentioned above).

Right, create the EG file again, and type in:

This is a Noddy file

Now create a new file called PROGRAM and we will take a look at how to create a program using the Noddy language. In order to let the computer know that we are issuing commands (as opposed to entering text) all Noddy commands must be preceded by an asterisk (*). Return to the PROGRAM file (assuming you are not already in it) and type in the following program. Remember that all commands must be entered in capitals if they are to be recognised as legitimate Noddy commands. Don't forget that you mustn't press <RET> until you have finished keying in the entire listing.

```
*DISPLAY EG.
*PAUSE
*RETURN
```

Press <RET> and presto ... nothing happens! In order to run a Noddy program we must return to BASIC. To do this simply press the CLS key followed by <RET>. Don't worry, in this instance it is perfectly safe to use this key, which only causes problems when you are actually in the process of creating a Noddy page. Now that we are in BASIC we can run the PROGRAM by using the PLOD command. Type in:

```
PLOD "PROGRAM"          <RET>
```

and the screen clears and displays the file called EG. Return to Noddy and recall the PROGRAM file by typing in PROGRAM and pressing <RET> so that we can take a look at the commands that were used to create this program. The first command was:

```
*DISPLAY EG.
```

This instruction, as you might expect, tells the computer to DISPLAY the file EG to the screen. You will notice that the filename was followed by a full stop (.). This must always be used when a filename is being referred to within a Noddy program, since it instructs the MTX to search for the file in question. If this punctuation is omitted the Symbol? error report is returned informing us that the computer was unable to complete its search.

The next command in our program was:

```
*PAUSE
```

The *PAUSE command stops the execution of a program for approximately one second each time it is used. So, if we had wanted to display this file for three seconds we would have had to key in:

```
*PAUSE
*PAUSE
*PAUSE
```

The final command that was used in this program was:

```
*RETURN
```

This command is always entered at the end of a Noddy program, instructing the MTX to RETURN to BASIC. If you omit this command the Symbol? error report will be displayed.

Once you have switched between BASIC and Noddy a few times you will begin to find it very tedious having to key in PLOD "xxx" to run the program and then NODDY to return to Noddy. So, you will be pleased to hear that both of these commands can be entered in a program format,thus:

```
10 PLOD "PROGRAM"
20 NODDY
```

Try RUNning this program and you will see that the Noddy PROGRAM displays the EG file for approximately one second and then returns to Noddy instead of BASIC. Leave this program in memory and just EDIT line 10 if you want to PLOD a different Noddy program. Apart from these considerations it is also very useful to be able to call upon the Noddy screens from within a BASIC program. particularly when you need to display impressive textual displays. Creating such screens from BASIC with the use of PRINT statements is unspeakably laborious.

One of the most impressive features of the Noddy language is the simplicity and speed at which a program can be entered. This is because all Noddy keywords (except DIR) can be abbreviated to their first letter and it is unnecessary to start a new line for each keyword. However, you do have to leave a space between the keyword and its parameter. For example, a space must be left between *DISPLAY and EG., and its omission will return a Symbol? error report.

This said, it is sensible to leave a few spaces between each command in order to make the program legible, otherwise it becomes very difficult to read a program when lines of symbols are joined together. Take a look at the listing below and you will see how difficult it is to read, despite the fact that it is a very short program.

***D EG.*P*P*P*R**

Create a new file called EG2 and then key in 'Please make an input and then press the <RET> key'. Now create another file called PROGRAM1 and key in the following:

```
*DISPLAY EG2.
*ENTER
*RETURN
```

The *ENTER command is not dissimilar to the BASIC keyword INPUT, since it halts the execution of a program until an input has been made. This facility can usefully be incorporated into a BASIC program. For example it could be used in conjunction with a set instructions saying something to the effect of 'Press the <RET> key to continue'.

If the truth were to be told an *ENTER command used in this way does seem rather a waste of time, but by using it in conjunction with the *IF command it has far greater possibilities. The *IF command is followed by two parameters. The first specifies the input that is required by the program and the second is called a pointer or label. The function of the second parameter (let's call it y) is to force the MTX to jump to the next y it finds. In order to avoid any confusion, the y that it is searching for is preceded by a ^. This prevents the MTX from attempting to PLOD the program from the wrong point. Create a new file called PROGRAM3 and then take a look at the following program which demonstrates this command:

```
      *DISPLAY EG2.
      *ENTER
      *IF YES,y
      *DISPLAY EG.
^y    *RETURN
```

This example *DISPLAYs the EG2 file and then waits for an input. The *IF command looks at the input to see whether it is the same as its first parameter (which in this case is YES). If they match it will then look at the second parameter (y) and scan the rest of the program for a matching ^y which it will locate before the *RETURN. If the input wasn't YES the MTX would have moved on to the next command and subsequently *DISPLAYed the file EG.

There are a couple of points to bear in mind when using the *IF statement. The first parameter cannot contain any spaces and the pointer flag (^y) must precede the asterisk (*) of a command.

The second parameter is referred to as a pointer because it is used to point to another part of the program; it doesn't have to be in lower case but it does help to clarify the listing if it is. As this is the most complex of the commands we have so far discussed, let's quickly recap how it works and what to look out for.

a) The first parameter is the desired input (x)
b) The second parameter is the pointer flag (y)
c) If the input matches the first parameter it searches for the pointer flag (^y)
d) The pointer flag (^y) must precede the asterisk of a command
e) If the input does not match the first parameter (x) it moves on to the next command
f) There should be no spaces contained in the first parameter

The next command we shall look at is *GOTO. This command is used to GO TO a previously defined Noddy program file. There is only one occasion when this command cannot be used, and that is when a filename contains a space. If you attempt to GOTO this type of file the program will crash and a 'No data'

error report returned. Create a new file called PROGRAM2 and enter the following listing:

```
DISPLAY EG2.
*ENTER
*GOTO PROGRAM.
```

You will notice that the *RETURN command has been omitted in this program. This is because the *GOTO statement passes control to another program which contains the necessary *RETURN command. However, the omission of the *RETURN statement is bad programming practice, so let's add it to the listing. The PROGRAM2 file should now look like this:

```
*DISPLAY EG2.
*ENTER
*GOTO PROGRAM.
*RETURN
```

Let's quickly run through this program. The program *DISPLAYs the file called EG2 and then awaits an INPUT (which is activated by <RET>). Once this has been done, the *GOTO tells the MTX to GO TO a file called PROGRAM. When the PROGRAM file is accessed it is then activated, thus displaying the file called EG. You will notice that the filename specified by the *GOTO command is followed by a fullstop to indicate to the MTX that it must search for a filename.

It is also possible to *GOTO the same program file, which has the effect of running a program from the beginning of the file. This kind of structure is sometimes used in conjunction with the *IF statement. Create a new file and call it PROG. Now key-in the following:

```
      *DISPLAY EG2.
      *ENTER
      *IF YES,y
      *GOTO PROG.
^y    *RETURN
```

This program will end once YES has been input otherwise it will keep on repeating itself. Even though the *GOTO command can be used quite effectively in this manner the *BRANCH command is more commonly used. The *BRANCH command is followed by one parameter which is a pointer flag that acts in the same way as the second parameter of the *IF statement. Create a new file called PROGRAM4 and we will rewrite the above program using the *BRANCH command.

```
^x    *DISPLAY EG2.
      *ENTER
      *IF YES,y
      *BRANCH x
^y    *RETURN
```

This program produces exactly the same display as the earlier *GOTO example but, as you can see, it is more efficient. Pointer flags are extremely useful because they enable us to create efficient and well structured programs. Before moving on to look at any new commands, let's construct a couple of programs which emphasise the value of pointer flags. Create a new file called PROGRAM5 and key-in the following program:

```
^x    *DISPLAY EG2.
      *ENTER
      *IF YES,y
      *IF NO,n
      *IF MAYBE,m
      *BRANCH x
^n    *BRANCH x
^m    *GOTO PROGRAM.
^y    *RETURN
```

This program asks you to make an input and with the use of the *IF and *BRANCH statements, decides what to do with it. If you *ENTER YES the program *RETURNs to BASIC (label x). If NO is *ENTERed (label n) the program BRANCHes to x and the prompt remains on the screen. If the input is *MAYBE the program GOes TO a file called PROGRAM (label m). Finally, if anything else is entered the program BRANCHes to X and re-displays the request for an input.

This type of program is very useful when you need to provide users with a menu of options. To demonstrate this concept create a new file called EG3 and key in the following:

Please press a number to perform the desired function:

1. **Return to BASIC**
2. **Read the EG file**
3. **Read the EG2 file**

Now create another program file and call it PROGRAM6.

```
^x  *DISPLAY EG3.
    *ENTER
    *IF 1,a
    *IF 2,b
    *IF 3,c
    *BRANCH x
^b  *DISPLAY EG.
    *PAUSE *PAUSE *PAUSE
    *BRANCH x
^c  *DISPLAY EG2.
    *ENTER
    *BRANCH x
^a  *RETURN
```

This program provides the user with a menu of options. If 1 is pressed the MTX will *RETURN to BASIC (label a). If 2 is pressed the file called EG will be displayed to the screen for approximately three seconds, followed by a reapperance of the menu. When 3 is pressed the EG2 file is displayed until an input has been made, when, once again, the menu is displayed in all its glory.

STACKING THE ODDS

All the files we have created so far have been stored in the MTX's memory in the form of a stack. This stack is very similar to a stack of plates or a pile of paper. When you want to look at a piece of paper you search through the pile until you locate the sheet required, and then remove it to examine its contents. Similarly, when you examine a Noddy file it must be removed from its position in the stack before it can appear on the screen.

The *STACK command enables the user to look at program files one after the other in any order. The specified filenames must be separated by commas and the final filename in the statement must be followed by a full stop in order to indicate a file. Thus:

***STACK PROGRAM, PROGRAM1, PROGRAM2.**

As the programs are arranged in the stack on a 'first in last out' basis, the MTX will start by locating PROGRAM2, then PROGRAM1 and finally PROGRAM. The *STACK command is used in conjunction with the *ADVANCE statement whose job is to instruct the computer to progress through the program stack, removing and executing each program in turn. However, an important point should be considered when using these commands: each program file will only be executed if the preceding file does not contain a *RETURN command, since such a statement will automatically *RETURN control to BASIC. In order to avoid such a situation we must substitute the *RETURN statements in our example programs with *ADVANCE statements. Let's move on and create a new file called EG3. Now enter 'PLEASE PRESS RETURN' and then change the PROGRAM2 file so that it *DISPLAYs EG3.

***DISPLAY EG3.**

(This is only a cosmetic modification which will enable you to distinguish between the PROGRAM1 and PROGRAM2 files). Now create a new file called PROGRAM7 and you will be able to see these commands in action.

***STACK PROGRAM7,PROGRAM,PROGRAM1,PROGRAM2.**
***ADVANCE**

(You will have to press the BReaK key to exit from this program.) Our latest example creates an eternal loop since none of the files contain *RETURN statements. Consequently the *ADVANCE statement will perpetually select and execute the programs stored in the stack.

You might be wondering why the above example included PROGRAM7 in the *STACK statement. The main reason for this is that its omission would generate a 'No call' error message because the PROGRAM file contains an *ADVANCE command. Under these circumstances the MTX will attempt to pull the next program off the stack only to discover that there is no program left to call.

A more elegant method of exiting this program could be facilitated by the inclusion of a *RETURN statement in the last program to be executed. So change the PROGRAM file into its original form by replacing the *ADVANCE statement with *RETURN. We must also make the modification to PROGRAM2 since it contains a *GOTO PROGRAM command and thus PROGRAM7 would be halted as soon as the *RETURN command had been executed. So, let's take out this *GOTO command altogether.

Now all we need to do is make a final cosmetic change to the PROGRAM7 file. As the PROGRAM *RETURNs to BASIC before the *ADVANCE command has a chance to call the PROGRAM7 file we might as well delete it. Thus the PROGRAM7 file should now look like this:

***STACK PROGRAM,PROGRAM1,PROGRAM2.**
***ADVANCE**

When we PLOD this program it will *RETURN to BASIC once all the current files have been executed.

It is also possible to extract a program file from the stack without actually executing it. This is acheived with the use of an *OFFSTACK statement. Let us suppose that we wanted to pull PROGRAM1 off the stack but we didn't want to RUN it. By inserting an *OFFSTACK command into the preceding program file (in this case PROGRAM2) we can bypass PROGRAM1.

Insert an *OFFSTACK command before the *ADVANCE command in the PROGRAM2 file.

***DISPLAY EG3.**
***ENTER**
***OFFSTACK**
***ADVANCE**

On PLODding this program you will see that the PROGRAM1 file has been ignored. Thus when the <RET> key is pressed (in PROGRAM2) the control of the program skips over the PROGRAM1 file and executes the PROGRAM file.

PRINTING NODDY FILES

When we want to print out a Noddy file we cannot make use of LLIST or LPRINT but must utilise the Noddy command *LIST. To print out the Noddy file EG you must simply open a file (let's call it LIST) and then enter in a one line program that reads:

***LIST EG.**

(Note the full stop.) However, before you can PLOD this program you will have to set up your printer so that it will accept a line length of 39 characters. The manual that comes with your printer will tell you how to alter line length.

LPRINT CHR$(27);"Q";CHR$(39);

will initialise the DMX 80 printer to print 39 characters per line.

Once the printer has been set up you are ready to PLOD the LIST program.

NODDY KEYWORDS

We have included the following summary of Noddy keywords as a quick reference guide. The majority of Noddy commands must be preceded by an asterisk (*) and all must be entered in capitals.

ADVANCE

SYNTAX: *ADVANCE
ABBREVIATION: *A

This command normally operates in conjunction with *STACK and is used to pull program files off the stack and execute them. Each program file that is accessed in this manner must replace its *RETURN command with an *ADVANCE statement if you require the next program that is extracted to be executed. If the *RETURN is retained, the MTX will immediately *RETURN to BASIC when it encounters the statement. If all of the *RETURN commands have been replaced with *ADVANCE the first program file following the *STACK command must contain the name of the program file itself. Thus, the following program file is called eg.

***STACK eg, fred, flo.**
***ADVANCE**

This program will extract the files flo, fred and eg from the stack and execute them in that order. Providing the MTX doesn't encounter a *RETURN command the program will run indefinitely and the only way of breaking into it is by using the BRK key.

The reason this example requires the inclusion of its own progam filename is because none of the extracted files contain *RETURN commands (they have been replaced by *ADVANCE statements). If it is omitted a 'No data' error message will be returned. This is because the fred file will attempt an *ADVANCE only to discover it has no place to go.

BRANCH

SYNTAX: *BRANCH x
ABBREVIATION: *B

This command is similar to the BASIC keyword GOTO and is used to *BRANCH to any point within a given program file. The statement's only parameter is referred to as a pointer or label since its role is to determine the point to which control is to be directed. When the MTX encounters the *BRANCH command it looks at its parameter and then scans subsequent lines of the program until it encounters the character specified. In order to avoid any confusion, the character which serves as the statement's destination flag (i.e. the character for which the *BRANCH statement searches) has the ^ symbol preceding it which ensures that the program *BRANCHes to the correct point.

```
^x   *DISPLAY FRED.
     *ENTER
     *IF YES,y
     *BRANCH x
^y   *RETURN
```

This program will *RETURN to BASIC if YES is *ENTERed, and any other input will cause the *BRANCH command to return control back to the beginning of the program.

DIR

SYNTAX: DIR
ABBREVIATION: NONE

DIR is short for DIRectory and is used to display all the Noddy files currently in the memory. It can only be entered as a direct command and is the only Noddy command that does not require a preceding asterisk. The statement must, however, be entered in capitals. A lower-case version of the command will be interpreted as a new file called 'dir', since the MTX makes a distinction between upper and lower-case characters.

DISPLAY

SYNTAX: *DISPLAY filename.
ABBREVIATION: *D

This command is used to DISPLAY another Noddy file to the screen. The filename must be followed by a full stop which tells the MTX that it must search for a file. If the full stop is omitted a 'Symbol?' error report will be returned, and this message will also occur if there is no space between the filename and the keyword.

```
*DISPLAY FRED.
*PAUSE
*RETURN
```

This program will *DISPLAY the file called FRED for approximately one second before *RETURNing control to BASIC. You will generate a 'No Data' error message if you attempt to *DISPLAY a non-existent file.

ENTER

SYNTAX: *ENTER
ABBREVIATION: *E

This command works on a similar principle to the BASIC keyword INPUT. It halts the execution of a program until an input has been made, and the input is entered by pressing the <RET> key. If you press the <RET> key without making any input, control will simply be passed to the following command.

```
*DISPLAY FRED.
*ENTER
*RETURN
```

Our example will *DISPLAY the file called FRED and then wait for an input followed by <RET> before *RETURNing to BASIC.

GOTO

SYNTAX: *GOTO filename.
ABBREVIATION: *G

This command passes control to a specified program file (it can be the file in which the statement appears).

The filename that follows the *GOTO must be followed by a full stop which tells the MTX to search for a file. If this full stop is omitted a 'Symbol?' error will be returned. This message will also be returned if you omit the space between the keyword and the filename. You cannot *GOTO a filename that contains a space, if you attempt this or if the file does not exist you will find yourself up against a 'No data' error message.

```
      *DISPLAY FRED.
      *ENTER
      *IF YES,y
      *GOTO FILE.
^y    *RETURN
```

This program will *DISPLAY a file called FRED and then wait for an input to be made. *IF the input is YES the program moves on to the *RETURN command. If any other input is made it *GOTOs the file called FILE.

IF

SYNTAX: *IF X,Y
ABBREVIATION: *I

This command is used in conjunction with the *ENTER command. It examines user-generated input from which it determines the nature of subsequent processing. The statement's first parameter contains the input response that will be judged True or False. If the input entered by the user of the program matches *IF's first parameter, the computer will search the program for the character flag defined by the statement's second parameter. To ensure that the MTX does not jump to the wrong point in the program the destination flag is always preceded by the ^ symbol. The *IF statement's second parameter is referred to as a pointer (or label), since it determines the point to which the program must jump if the input test is judged True. If the input does not match the *IF statement's first parameter the control of the program is passed onto the command that follows the *IF statement.

```
      *DISPLAY FRED.
      *ENTER
      *IF YES,y
      *GOTO PROG.
^y    *RETURN
```

This program *DISPLAYs the file called FRED and then waits for an input to be made. The *IF statement basically says 'IF the input was YES then go to ^y'. In other words, if the input is YES the program will *RETURN to BASIC. If anything else is *ENTERed control passes on to the command following the *IF statement (i.e. *GOes TO a file called PROG).

There are a couple of rules to remember when using the *IF statement. The first parameter can contain no spaces and the pointer flag (^y) must precede the asterisk (*) of a command.

LIST

SYNTAX: *LIST filename.
ABBREVIATION: *L

This command is used to print out a program. However, before you can print a program out you will have to set up the printer to accept a line length of 39 characters.

***LIST FRED.**

The above statement will print out the file called FRED. (Don't forget to use the full stop after the filename.)

OFFSTACK

SYNTAX: *OFFSTACK
ABBREVIATION: *O

This command tells the computer to extract but not execute the next program from the stack. *OFFSTACK is used in conjunction with *STACK and *ADVANCE statements.

PAUSE

SYNTAX: *PAUSE
ABBREVIATION: *P

Each time this command is used it *PAUSEs a program for approximately one second. In other words, if you want to *PAUSE a program for three seconds you must enter the command three times. *PAUSE is usually used in conjunction with the *DISPLAY command in order to give users a chance to read the file that is being *DISPLAYed to the screen.

```
*DISPLAY FRED.
*PAUSE
*PAUSE
*PAUSE
*RETURN
```

This program will *DISPLAY the file called FRED for approximately three seconds before *RETURNing to BASIC.

RETURN

SYNTAX: *RETURN
ABBREVIATION: *R

This command must always be included in a Noddy program since its function is to *RETURN control to BASIC. If it is omitted a 'Symbol?' error report will be returned. If you want to return to Noddy after PLODing a program you must PLOD the program within a BASIC program and include the return to Noddy command in the following type of statement:

```
10 PLOD "FRED"
20 NODDY
```

This program will PLOD the file called FRED and will then return to NODDY once this operation has been concluded.

STACK

SYNTAX: *STACK (own),filename,filename....,filename.
ABBREVIATION: *S

This command enables you to execute multiple programs within a single program, where each filename is separated by a comma except the last which must be followed by a full stop. *STACK is used in conjunction with the *ADVANCE command which extracts programs from the stack in the order specified by the *STACK command and then executes them.

The syntax above shows that sometimes the first filename following *STACK is that of the file in which the command appears. This format is only used when there are no *RETURN commands included in any of the extracted programs (for further explanation of this refer to the *STACK entry).

***STACK fred, freda, jack, john**

will first extract the file called john, then jack, followed by freda and finally fred.

N.B. A good way of remembering the Noddy Keywords is this association:

P	**PLOD**	**B**	**BRANCH**	**E**	**ENTER**
L	**LIST**	**I**	**IF**	**A**	**ADVANCE**
O	**OFFSTACK**	**G**	**GOTO**	**R**	**RETURN**
D	**DISPLAY**			**S**	**STACK**

CHAPTER 12 : THE MTX ASSEMBLER

WHAT IS ASSEMBLER?

Having discussed BASIC at some length and outlined the essentials of Noddy, the time has come to introduce the programming facilities offered by the MTX Assembler. The principles of Assembly language programming are extremely complex; a thorough introduction to the subject deserves a book to itself, and thus well beyond the scope of this manual. However, since the MTX's Assembler facilities are one of the machine's most powerful features, no introduction would be complete without a brief outline of the language it accesses and the reasons why all BASIC programmers should consider learning how to harness its power.

English is referred to as a high level language because it utilises very complex conceptual and syntactical structures which, precisely because of the scope of expression offered by such complexity, enable us to issue sophisticated instructions or statements in very few words. At the other end of the communication spectrum, there are low level languages which require step by step instructions, with each element of a process represented by a signifier from the language in question, before any intent can be unambiguously expressed.

Although computer languages are far less complex than the languages with which humans communicate, PASCAL, ALGOL, FORTRAN, COBOL and BASIC are also considered 'high level' because, like English, they facilitate 'conceptual' rather than 'elemental' communication. The following analogy should help to clarify the difference between high and low level languages. Consider the English instruction "go and make a cup of tea". As soon as you hear this you understand what is expected of you. However, a lower level language would require more instructions. For example:

```
BOIL KETTLE
PUT TEA IN POT
POUR WATER INTO TEAPOT
POUR TEA
GIVE TO DRINKERS
```

Even when a problem presented to the computer has been broken down in this manner, the machine will still be unable to directly understand any stage of the process. This is because micros make use of 'machine code', the lowest possible level of language. So, moving down the language levels, the machine code version of the tea analogy becomes:

```
IDENTIFY KETTLE
IDENTIFY TAP
MOVE KETTLE TO TAP
TURN TAP ON:   IS KETTLE FULL?
                  IF NOT GOTO IS KETTLE FULL
TAP OFF
MOVE KETTLE TO STOVE
IDENTIFY MATCH
IDENTIFY GAS TAP
STRIKE MATCH
ETC,ETC.
```

As you can see, the process of simplifying the components of a problem generates an enormous list of instructions and decisions. The level of simplification required by a language is determined by the simplification of the processor in question - i.e. a Z80 chip (as with the MTX) or a human brain.

In the same way humans automatically break down the apparently simple instruction "make a cup of tea" into conceptual components, computers break down a high level language, such as BASIC, into simple instructions which can be executed by the microprocessor. When you enter a program in BASIC, each element of every instruction is translated into machine code by the 'interpreter' which resides in the MTX's BASIC ROM (Read Only Memory).

Like BASIC, machine code can be entered directly from the keyboard. The advantage of writing programs in machine code is that they are executed considerably faster than their BASIC equivalents. This is because the MTX doesn't need to waste any time interpreting each instruction before it can be executed.

The machine code used by all digital computers is made up entirely of 0's and 1's. These zeroes and ones represent either voltage low (0), or voltage high (1), and are used to regulate the electronic circuitry at the heart of every microcomputing operation.

Unsurprisingly, the human brain tends to find sequences of zeroes and ones very confusing. Thankfully, the MTX is blessed with an instruction based language facility called an Assembler, which can be used to make machine code more meaningful to humans.

Assembly code uses mnemonic (aids to memory) instructions and the MTX uses the Z80 Assembler language. The sequence of zeroes and ones below is an example of a typical machine code instruction (Z80):

10000110

(You'll be thrilled to learn that this particular instruction 'adds' the contents of the address held in the HL register to the Accumulator.) The corresponding assembly code instruction is:

ADD A,(HL)

Even without understanding the operation performed by this instruction, it should be clear that the mnemonic instruction (ADD A,(HL)) is a lot more meaningful than its machine code equivalent (10000110).

ADVANTAGES AND DISADVANTAGES

It tends to be both easier and faster to write bug-free programs using a high level language such as BASIC, than attempting the same task using a low level assembly code language. BASIC listings are shorter, more readable and to a great extent machine independent - which means a program can work on different computers with minimal alteration.

However, although the commands used by high level languages are more powerful (because they've been designed to be user friendly rather than machine friendly), there are many unnecessary instructions and transfers made to the micro's memory. This drastically slows down the running speed of a program and also uses a great deal of valuable memory space.

Another disadvantage of BASIC is that it's an 'interpreted' language; when a program is RUN each line has to be translated into machine code before it can be executed. Most other high level languages are compiled, which means that the whole program is translated into machine code before it's RUN. This not only dramatically increases the speed of execution, but also means that any line encountered more than once will not have to be 'reinterpreted'! When the MTX encounters a BASIC FOR...NEXT loop structure, it will have to 'reinterpret' the statements within the loop on each pass.

The table below shows the amount of time it would take each different language to process the same program, and will hopefully clarify the inefficiency of high level languages:

ASSEMBLER	10 Seconds
COMPILED LANG.	25/30 Seconds
INTERPRETED LANG	200/300 Seconds

So the main advantage of Assembler/machine-code is the speed at which it activates commands. Assembler programming also has the hidden advantage of helping us understand the mechanics of data processing, thus adding a practical dimension to the theory of efficient coding.

Since assembly language coding is such a time-consuming business, most programmers avoid writing complete programs in the language. Instead, selected routines are coded with the aid of the Assembler, which are called from a standard BASIC program. Let's take a look at how the MTX Assembler facilities can

be accessed from BASIC.

GETTING INTO ASSEMBLER ON THE MTX

Having hopefully aroused the interest of BASIC programmers unfamiliar with the world of machine-code programming, the time has come to send them away in search of a Z80 tutor. As we have established, the object of this section is to offer an introduction to the use of the MTX Assembler as a tool. To achieve this objective within the space available, we must assume a working knowledge of the coding facilities to which the Assembler provides access. In spite of appearances to the contrary, the intent of this preamble is to seduce those of you unfamiliar with Z80 programming into taking an interest in this MTX facility. Your computer's Assembler is one of the machine's most advanced features and, if you take the trouble to master the language it utilises, will prove of enduring value.

The immediate appeal of the MTX Assembler when compared with such facilities on other micros, is that it is incredibly easy to use. It is called from BASIC as a straightforward 'in line' assembler and only the machine-executable object code is stored in memory. Readable assembler SOURCE code is generated by disassembling the object code in question and inserting the relevent text and labels stored in tables below the object code.

Before we go on to explain the simple command sequences which access the facility, there is an important point to be stressed which must be borne in mind whenever programming assembler routines. Since the code you create will be stored as a BASIC line, the actual location of your code will change if the BASIC program is modified or extended. This location change will sometimes result in a final version of a program that will not RUN. However, it is relatively simple to resolve this problem since the code can be reassembled by re-entering the assembler at the code line before exiting once again.

Having resolved this potential source of grief at the outset, let's press on in a more positive vein and examine the mechanics of accessing the Assembler.

WRITING IN ASSEMBLER

As we have already explained, MTX assembler code is held in a BASIC line. Consequently our first task is to let the computer know the line number in which the code is to be held. This task is performed by the ASSEM statement entered as a direct command. For example:

ASSEM 10

defines line10 as an assembler code line. A statement of this type is entered by pressing the <RET> key, which will clear the screen and generate the:

Assemble>

prompt in the bottom left-hand corner of the screen. You are now in the assemble mode! The effect of this process is to neutralise the action of BASIC commands like LIST and RUN. Your MTX is now anticipating one of the following assembler instructions:

C - Clear screen	**P - Print to printer**
L - List to screen	**T - Top of program**
E - Edit	**Insert (by default)**

Having established a start point for our routine and itemised the commands which facilitate its creation, let's take a look at how the MTX Assembler enables such code to be written.

The first step in the creation of your assembly code is to simply press the <RET> key once again. As the command list above reveals, this process accesses the INSERT mode (which is entered by default). Once this is accomplished an 'Insert>' prompt will appear at the bottom of the screen. A four digit hexadecimal number and flashing cursor will be displayed a few lines above this prompt. The hex value represents the

memory location to which the next instruction will be assigned. Finally, to the right of the cursor will appear the instruction RET. So at this stage our screen display will look something like this:

Hex addr no. RET
Insert>

At this point things become even simpler! By making use of the EOL key on the numeric keypad you can erase the RET instruction and enter the first instruction of your routine. Once this has been keyed-in and edited, it can be entered by pressing the <RET> key. This will cause the instruction in question to disappear and the hex value to be incremented, indicating the location that will be assigned to the next instruction and the instruction currently at that location.

Having completed your routine, press the CLS key on the numeric keypad followed by <RET>. The screen will be cleared and the Assemble> prompt returned.

You are now faced with a number of options. It could be that you require a listing of the completed routine to be displayed on the screen. This can be achieved by pressing 'T' followed by <RET>. This sequence will set the program location pointer to the first instruction of the assembler routine. If you then press 'L' followed by <RET> the routine will be listed on the screen.

When there is more code than will fit on a single screen, the first screenful will be displayed followed by a bell prompt which indicates there is more code to follow. The next screenful can be displayed by pressing any key on the alphanumeric keyboard.

If you want to stop the listing, simply press the BRK key on the numeric keypad. Once the listing process has been completed, the Assemble> prompt will re-appear at the bottom of the screen, indicating that the MTX is awaiting further instructions.

So how can we edit our routines? Well, let's take a look at a simple example. In the following routine, the data stored in the five bytes starting at DATA are transferred to the five bytes starting at COPY:

```
8007            LD HL,DATA
800A            LD DE,COPY
800D            LD BC,5
8010            LDIR
8012            RET
8013 DATA:      DB 12,#34,"LOW"
8018 COPY:      DS 5
801D            RET
801E            RET

Symbols:
DATA     8013     COPY     8018
```

Now suppose we decide that we only want to transfer the first two bytes of data. This would require a change in the instruction at 800D. We will have to make use of the assembler's editing facilities to make the following modification:

800D LD BC,2

To access the edit mode simply enter 'E' followed by the location holding the instruction to be modified. In this case:

E #800D

Note the hex number must be preceded by the hash (#) symbol (obtained by entering a SHIFTed 3 from the main keyboard). Once this has been keyed-in, press the <RET> key. This operation will cause the specified instruction to be displayed. To make the required change, move the cursor along the line with the arrow keys and when it covers the '5', simply press the '2'. This will change the action of the instruction.

Once the editing is complete press <RET> to consign it safely to memory. This done, the MTX will display the next line of the routine for editing.

To exit the edit mode, press CLS <RET> and the assembler will be ready to receive further directives.

At this point it's worth mentioning that a routine's labels provide an alternative means of accessing particular instructions. For example, in the routine we have just modified we could make a change to the value at 8013 by utilising the label associated with this location. By entering:

E DATA <RET>

we can cause the appropriate line to be displayed for editing.

When you are finally satisfied with the fruits of your labour, you can exit the assembler and return to BASIC by pressing CLS <RET>. If you now list your BASIC program in the usual manner (i.e. entering L. or LIST), you'll notice that the MTX has inserted the line number (at which you specified the storage of the assembler routine), followed by the label CODE. CODE is not a BASIC statement, but simply acts as a tag which indicates the location of an assembler routine. Following the CODE line, you'll find the full assembler listing, after which the remainder of the BASIC code will appear.

At this point it's worth reiterating the warning given at the beginning of this section. The bulk of machine-code routines are location dependent and having completed an assemble routine any subsequent editing of BASIC code will relocate the assembler instructions. So your routines will almost certainly have to be reassembled after a BASIC edit session. We'll repeat the outline of the sequence required for such an operation. Once BASIC editing has been completed you must enter:

ASSEM ln <RET>

where ln is the line number at which your code is stored. This done, you complete the reassembly process by exiting in the usual manner (i.e. pressing CLS followed by <RET>). This process will achieve a reassembly of your code at its (hopefully) final resting place in the completed program. One obvious precaution you could take, which (in many cases) will enable you to avoid this problem altogether, is to place your CODE line(s) as near the start of the BASIC program as possible.

Having outlined the use of the MTX Assembler as a programming tool, for ease of reference it is probably worth reiterating the syntax and action of each of its commands. However, before so doing, a few words about the number systems accepted by the facility.

The MTX Assembler will accept values expressed in either a decimal or hexadecimal format, assuming the former by default. A number will only be interpreted as a hexadecimal representation if it's prefixed by a hash (#) symbol (SHIFT and 3). Words are stored according to the time- honoured Z80 convention - low-byte first, then high-byte.

ASSEMBLER COMMAND OUTLINE

As we have seen, the MTX Assembler uses only six commands, one of which (Insert) is assumed by default.

LIST

SYNTAX : L <hex/dec loc> or <label>

If entered without a parameter, L will list the entire assembler routine from the current position of the location pointer. If a location or label is incorporated in the statement's format, the listing commences from the point specifed. The command only displays a screenful of code at a time. If the bell prompt sounds this indicates that more code follows and any key-press (on the alphanumeric keyboard) causes the next screenful to be displayed.

EDIT

SYNTAX : E <hex/dec loc> or <label>

Accesses the assembler's edit mode. If E is entered without a parameter, the instruction displayed on the screen for editing will be determined by the position of the location pointer. When a location or label is included the instruction stored at the specified position will be displayed. The arrow keys facilitate movement over the current edit line.

To delete a line of assembly code, enter the edit mode at the required line and erase the instruction leaving the HEX number. When <RET> is pressed, the instruction will be deleted and the remainder of the code adjusted to fill the free space.

INSERT (default mode)

SYNTAX : <hex/dec loc> or <label>

To enter the Insert mode it's only necessary to press the <RET> key in response to the Assemble> prompt. This produces the display of the current code location into which an instruction can be inserted. The line will take the form of a hex number denoting the next location at which an instruction can be stored. If no instruction has been previously assigned to the location, the hex value is automatically followed by RET which must be deleted. This can be achieved by: pressing EOL <RET> (after an instruction is inserted) or, alternatively, overwritten by an instruction (if it was long enough).

PRINT

SYNTAX : P <hex/dec loc> or <label>

If entered without a parameter, P will print the entire assembler ROUTINE to the printer from the current position of the location pointer. If a location or label is incorporated in the statements format, the print commences from the point specified.

CLEAR

SYNTAX : C
C followed by <RET> will clear the screen.

TOP OF MEMORY

SYNTAX : T

This final assembler command requires no parameters. Its execution returns the program line location pointer to the start point of the routine currently being assembled.

To exit a given mode, simply press the CLS key on the numeric keypad followed by <RET>. This will return the Assemble> prompt.

To exit the assembler, return to the Assemble> prompt and enter CLS followed by <RET>. This will return the MTX to BASIC.

This concludes our introduction to the method of accessing the MTX Assembler and the commands by which it is controlled. Before moving on to outline the power of the MTX's PANEL facility, the following notes should be considered when using Z80 code to create programs with the MTX Assembler.

ASSEMBLER NOTES

The execution of machine code:

The task of executing machine-code routines can be accomplished in one of two ways. By now you should be familiar with the simplest solution to the problem, since it is implicit in our introduction to the MTX Assembler. If the machine-code routine is located in such a way that it interrupts the flow of the BASIC program, the computer will be forced to treat it as a hurdle which must be overcome (or processed!) before the remaining BASIC code can be executed. In other words, the only way the MTX can pass a strategically placed machine-code routine is by processing it (in anticipation of the RET statement that will return control to BASIC).

When a given routine is required more than once in a program, the simplest approach is to incorporate the codeline in a BASIC subroutine which can be called as and when required. This enables the codeline to be placed near the beginning of the BASIC program, thus reducing the number of occasions the 're-assembling' process is required as a consequence of BASIC editing.

The second method of accessing a machine-code routine is by means of the USR statement. Although the need for such a command is considerably diminished by the facilities offered by the MTX Assembler, its implementation is quite straightforward. When the computer encounters a statement such as:

100 LET X=USR(32775)

control will pass control to the machine-code routine at the location specified by the statement's argument. Once it has been executed the control returns to BASIC and USR holds the current value of the BC register pair. This can now be accessed via its variable (in the case of the example above we could PRINT X), or by using the format:

120 PRINT USR(32775)

Pseudo operations:

Although they bear more than a passing resemblance to Z80 instruction mnemonics, pseudo operations are, in fact, a means of reserving space in memory and defining its contents. Only three such operations are required by the MTX Assembler - DB, DW and DS. The following section outlines the role and syntax of each instruction.

OPERATION:DB - DEFINE BYTE

SYNTAX: <label:>DB<hex/dec>and/or "<string>" and/or <label>

Defines the contents of the byte of memory specified by the instruction's argument. Its parameters can be numeric (expressed as either decimal or hexadecimal values), alphanumeric, or the low byte of a label. (It should be noted that although the use of a label will generate an out of range error, but by ignoring the message the low byte of the label can caused to remain in memory.)

2.OPERATION:DW - DEFINE WORD

SYNTAX: <label:>DW<hex/dec> and/or <label>

This instruction assigns the value specified by its argument to a word (i.e. two bytes). The assigned value is stored high byte first, and is thus compatable with Z80 word instructions.

3.OPERATION:DS - DEFINE SPACE

SYNTAX:<label:>DS <dec/hex>

Instruction which reserves the quantity of memory space (between 0-255 bytes) specified by its argument.

The following examples should help to clarify the various format options applicable to such operations:

```
DATA:     DB 10,#20,"HIGH"
          DB "SCORE"
JMPTAB:   DW START,START1,0,#F0E3
          DW HIT,WIN
BUFFER:   DS 50
          DS #40
```

Comment insertion:

The addition of comments to MTX assembler instructions takes the standard format. In other words, they must be prefixed by a semi colon (;) and are delimited by the end of line. The inclusion of comments after non-executable lines requires the use of NOP. For example:

```
          NOP ;Routine to show comments
          NOP
ROUTE:    INC A
          RET
```

Listing, Loading and Saving:

Because of the way assembler code is stored on the MTX (i.e. as a BASIC line), the listing, saving and loading of assembler code uses exactly the same methods as those employed when carrying out the equivalent operations for BASIC code.

Non-standard features:

Programmers with previous experience of assembler facilities will have noted a number of significant differences between the MTX Assembler and other assemblers. Such distinctions are undoubtedly positive, since they centre around the fact that the facility has done away with the need to specify the origin of object code or provide endless lists of assembler directives. Such economies have resulted in an assembler that is unusually easy to use and relatively fast (approximately 2-3 seconds for an 8K program).

Finally, it should be noted that although the Z80 instructions usually associated with loading the stack-pointer with HL,IX and IY are not directly available from the assembler, they can be accessed as follows:

Normally:	Use instead:
LD SP,HL	DB #F9
LD SP,IX	DB #DD,#F9
LD SP,IY	DB #FD,#F9

Restart for screen output:

All screen output ROM calls are accomplished by means of RST 10 calls. The effects of a given call are determined by the data following RST 10. Since the data in question is actually stored in the path of the program, you may initially find this approach a little confusing, but in fact it's simplicity itself! This can hopefully be demonstrated by getting down to specifics.

1.Writing ASCII to the screen

RST 10 to pass representations of registers B and C to the screen:

```
8007            LD BC,"TM"
800A            RST 10
800B            DB 192
800C            RET
```

Whilst this demonstrates the principle of the operation, it's not particularly interesting. The next routine is slightly more significant:

```
8007            LD E,9
8009            LD HL,DATA
800C LOOP:      LD A,(HL)
800D            LD B,0
800F            LD C,A
8010            RST 10
8011            DB 192      ;write BC token
8012            INC HL
8013            DEC E
8014            JR NZ,LOOP
8016            RET
8017 DATA:      DB " MEMOTECH"
8020            RET

Symbols:
DATA    8017    LOOP    800C
```

2. Messages to the Screen

The format which follows shows how we can send a complete string and hence avoid the inherent complications of the last routine:

```
8007            RST 10
8008            DB #8C,"MEMOTECH LTD"
8015            RET
```

The byte following the RST 10 instruction takes the following form:

7	6	5	4	3	2	1	0
1	0	c	←		n		→

in which bit 5 indicates that the routine should go on to interpret the data following the instruction, and n is the number of bytes in the string.

3.VS and RST 10

The format for the Virtual Screen instruction byte takes the following form:

7	6	5	4	3	2	1	0
0	1	c	*	cls	←	n	→

c is the continuation bit; n the selected Virtual Screen; cls the screen clear option.* indicates that the bit can assume any value and have no operational effect when used in this mode.

4.Single Byte to Screen

This call facilitates a single byte screen transfer. It takes the following form:

7	6	5	4	3	2	1	0
0	0	c	*	*	*	*	*

c=continuation bit:*=no operational effect in this mode, whatever its value. The effect of this call is demonstrated by our next example which makes use of RST 10 and CALL #79 (keyboard input). The routine enables its user to ramble around the screen creating a display which echoes keyboard input.

```
8007 START:   CALL #79
800A          JR Z,START
800C          LD C,A
800D          LD B,0
800F          RST 10
8010          DB 192
8011          CP 13
8013          JR NZ,START
8015          RET

Symbols:
START    8007
```

5.RST 10 and Control Codes

One of the most powerful RST 10 calls combines the instruction with the MTX's Control Codes. These are represented by the first thirty-two codes of the ASCII set, and can be considered "invisible" in that they cannot be PRINTed to the screen.

Finally, the following chart lists the commands that can be accessed via RST 10, along with their ASCII codes:

ASCII CODE	FUNCTION
1	PLOT X,Y
2	LINE X1,Y1,X2,Y2
3	CURSOR X,Y
7	BELL
10	LINE FEED CURSOR DOWN
11	VERTICAL TAB
12	CLS/HOME
13	CRGE RETURN
14	CTLSPR P,X
15	GENPAT P,N,D0,D1,D2,D3,D4,D5,D6,D7
16	COLOUR P,N
17	ADJSPR P,N,V
18	SPRITE N,P,XP,YP,XS,YS,COL
19	MOVSPR P,N,D
20	VIEW DIR,DIS
21	INSERT KEY
22	DELETE KEY
23	BACK TAB
25	TAB KEY
26	HOME KEY
27,65	ATTR P,STATE
27,89	CRVS,N,T,X,Y,W,H,S
27,90	VS N
27,67	GR$ X,Y,B (result in work space)

THE PANEL FACILITY

Even with only a rudimentary grasp of the principles of machine-code programming, the PANEL facility will prove to be a godsend to your programming development. After you've mastered your computer's more conventional features, experimentation with PANEL's potential will almost certainly persuade you to regard it as the dark horse amongst the MTX's features!

PANEL is normally entered from BASIC as a direct command. Its execution forces the computer to draw aside the veils that cloak the inner workings of the MTX by offering a window which displays the computer's memory and registers. Thus PANEL allows the direct examination of the Z80 registers and sections of memory. The obvious value of this feature is that it offers MTX programmers unrivalled debugging facilities, providing a means of stepping through a flawed creation instruction by instruction! Let's take a look at the options open to us when the PANEL mode is entered.

Having made it clear that the mode is normally accessed from BASIC by the use of PANEL as a direct command, it's worth mentioning that it can also be entered from within a program as a BASIC statement or in an assembly routine using RST 38.

The top right of the screen displays the Z80 registers, whilst the bottom of the screen presents a section of the MTX's memory. Having entered the PANEL mode, standard keyboard response is neutralied and the MTX will expect the single keypress input corresponding to the facility commands listed below. Before looking at the action of these commands, let's see how cursor control is accessed.

The PANEL display presents two cursors (>), one for the registers and one for the memory block.

<RET>	moves the memory cursor forward
↑	moves the memory cursor up
(down cursor)	moves memory cursor down
-	moves memory cursor back
.	moves the register cursor

The command list which follows should give you a hint of the power and flexibility of PANEL. One of the few valid 'obvious truths' that are applied to the over-mystified world of machine code is that it is indisputably a pain to debug. There's no point pretending that the MTX PANEL will answer all your problems, but intelligent experimentation facility will undoubtedly enable you to reduce the tedium coefficient normally associated with the fault-finding process!

As we have already established, all PANEL commands are accessed via a single keypress - which in most cases is the first letter of the full command name. Once a command letter has been entered, the selected operation will either be immediately executed (as in the case of Clear), or else produce a command prompt at the bottom the screen. The initial evocation of a PANEL command does not require <RET>.
On obtaining the command prompt, you must key-in the hex number(s) of the location(s) appropriate to the selected command, which is then executed with <RET>. The only exception is the BASIC prompt, which will return to the BASIC mode in response to a 'Y' input.

In the majority of cases, PANEL commands which generate a prompt should never be entered without parameters. This is particularly true of commands like GO and MOVE, the execution of which, if ill-considered, can often have tragic effects on your code. It should always be borne in mind that the power of the PANEL mode is precisely that it enables you to tamper with your creations at source, with all the dangers that such an innovation implies. In short, experiment with test material before meddling with (the unsaved version of) anything you value!

With one exception, the execution of a PANEL command will produce an empty prompt line which indicates that the facility is awaiting further instructions. However, DISPLAY will continue to present the locations indicated by the memory cursor until <RET> is preceded by invalid (i.e. non-hex) input.

THE PANEL COMMANDS

All commands are accessed by a single keypress, normally the first letter of the command name.

BASIC

Facilitates exit of PANEL mode and a return to BASIC. Generates an 'Exit?' prompt, to which anything other than 'Y' <RET> will ensure a perpetuation of the current mode.

CLEAR

Erases displays generated by the LIST command, and returns the screen to the display format encountered on entering PANEL.

DISPLAY hex loc.

Displays the location specified by its parameter (hex loc.). If the command is entered without a parameter, PANEL will display the location currently indicated by the memory block cursor. When DISPLAY is executed, PANEL's 'memory window' (i.e. lower display) will re-form to present the memory area around the location specified. The location in question and its contents are presented at the bottom of the PANEL screen, followed by the flashing cursor. <RET> without additional input will cause the memory cursor to move on to the next location, which will then replace the display of the location that was initially specified. The contents of the current location line can be altered if a new (hex) value is entered (followed by <RET>). PANEL will continue to present consecutive locations until a non-hex value is entered.

If display is followed by '.' memory at the value of the register pointed to by the register cursor will be displayed.

GO hex loc1 (TO) hex loc2

Command which enables a program to be RUN, the start and end points of which are specified by hex loc1 and hex loc2 respectively. When GO is accessed, its command prompt appears, indicating that the program's start point (hex loc1) is required. Once this has been entered followed by <RET>, the TO prompt will appear, and the program's end- point location (hex loc2) must be entered, enabling the execution of the specified code.

ASCII/HEX DISPLAY

Unlike most of the other PANEL commands, the ASCII/HEX toggle is not accessed by its initial letter, but by entering I. It requires no parameters, since its function is simply to determine whether hex or ASCII representations are displayed by the memory window and the location line produced by DISPLAY. The action of the command is to replace hex with ASCII, and vice versa.

LIST hex loc

List code from location specified by its parameter. If hex loc is omitted, the display generated will be determined by the progran pointer's current position. A LIST display is created to the left of the register block, and can be cleared by means of the CLEAR command.

If list is followed by '.' code will be listed from the current value of the program counter.

MOVE hex loc1 (END) hex loc2 TO hex loc3

The command sequence initiated by MOVE allows a defined section of memory to be moved to a specified location. The first command prompt (Move>) requires the start point of the memory area to be specified (hex loc1). When this has been entered, the End> prompt appears, requiring the input of the end-point location (hex loc2). Finally, the TO> prompt requests to location (hex loc3) to which the section of memory is to be moved.

REGISTER hex

Command which allows you to change the register currently indicated by the register cursor according the the value established by the statement's parameter (hex). If the command is entered without a parameter, it cannot be executed and PANEL simply awaits subsequent instructions.

SINGLE STEP

This command is entered without parameters, and facilitates the execution of the command at which the program counter is currently located.

TRACE

This command is entered without parameters. Its action is exactly the same as that of SINGLE STEP, except that calls are treated as a single instruction.

REGISTER TOGGLE

Like the ASCII/HEX toggle, the command that allows access to the top display's alternate register uses an unrelated command letter - X. Entered without parameters, its action enables PANEL to display the registers using one of two register sets.

GLOSSARY OF TERMS

Absolute Address	Information or data held in a computer is found by the address of its location. In machine code programs, the number defining an address is called an absolute address.
AC	Alternating Current.
Access Time	How long it takes to reference an item in memory.
Accumulator	A type of register.
ADC	Analogue to digital converter. Converts analogue signals into digital signals, would you believe! There are also digital to analogue converters, which work in the opposite direction.
Address	Each memory location has an address, used to find data or a program instruction.
Algorithm	A set of steps for performing a task
Alphanumeric	Numbers, letters and sometimes other things
Array	An arranged set of values linked by some kind of logical relationship. Each element in an array has a unique reference.
ASCII	American Standard Code for Information Interchange. Pronounced 'Askey', it's just a way of representing alphanumeric characters in binary. Difficult to get away from this one, it crops up all over the place
Assembler	A programming language one step away from the zeros and ones the computer understands and uses. Assembly code is the coding for a program written in assembler.
Backup	When things go wrong, if you haven't got one, you're in trouble.
BASIC	The Beginner s All-Purpose Symbolic Instruction Code.
Baud Rate	Number of bits per second transmitted along a line.
BCD	Binary Coded Decimal. A way of expressing decimal numbers using bits. Uses four binary bits for each decimal number.
Benchmark	A standard set of tests for seeing how fast a computer can perform. Used mainly in comparing one computer with its rivals
Binary	Number system using only two digits, 1 and 0.
BIT	Binary digit. Either a zero or a one, it is the basic unit of information storage.
Boolean Algebra	Set of logical instructions written using algebra, with an answer either TRUE or FALSE
Bootstrap	A set of instructions held permanently in the computer which have to be loaded before the computer can load programs
Branch	In programming terms, a branch is a part of a program where a decision is made and the program flow is transferred depending on the result. This is a conditional branch. An unconditional branch is something like the GOTO statement, where the program control jumps somewhere else without a decision being made.
Buffer	Somewhere data is stored temporarily, until the CPU is ready to process it. Also used to allow one part of the computer to work at a different speed from another part.
Bug	We all get these, so don t worry. A software error.
Bus	A set of connections which allow a route around the computer for signals.
Byte	A set of bits, the smallest unit that means anything. One byte is normally represented by 8 bits, and represents a character or number.
Centronics	A manufacturer of printers. Very popular. Lucky you've got a centronics type interface
Chip	This is what most people call an integrated circuit. It's a tiny piece of silicon, and the bread and butter of computers. (No jokes please.)
Command	An instruction to the computer to tell it to do something.
Compiler	Translates source code into object code.
Constant	Something (either a number, or a string) which doesn t change
CP/M	Stands for Control Program/Monitor. A widely used and well recognised operating system which makes available to you a wealth of software packages
CPU	The Central Processing Unit is a complex chip where all the logical and arithmetic operations are carried out. It's your computers brain
Crash	Something that happens to programs. When a program crashes it s because the computer has encountered an instruction which has totally confused it, so instead of getting an error message you usually get nothing, or lots of rubbish displayed on the screen
Cursor	The cursor tells you where the character you are about to type will appear. It's the blob on the screen that s about the size of an ordinary character
Data	Data is information which can be processed, stored or produced by a computer
DC	Stands for Direct Current. A constant voltage

Debug	The identification and removal of errors from a program.
Disc	An L.P. shaped plate covered in magnetic material which can store information or data on its concentnc tracks. Discs have a fast access time, because the read/write head can position itself quickly over the required data without having to read all the preceding storage area
Dump	To make a backup of e section of memory by printing it, or sending it to a backing store. to give a security copy usually.
Edit	To change data from what it was to whet you want it to be
Emulator	Software which enables one computer to duplicate the instruction set of another.
EOF	Stands for End Of File.
EPROM	Erasable, Programmable Read-Only Memory.
Execute	The carrying-out of a program or single instruction.
File	A file is a block of data organised so thet it can be stored and retrieved es required. Files always have names.
Flag	An indicator used to indicate something about data. For instance the Z80 CPU hes a flag which tells you whether the last operation performed resulted in zero or non zero.
Floppy Disc	Cheap, flexible store for data.
Flowchert	A graphic wey of representing the order of a set of events.
Gate	A single logic function.
Gigo	Garbage in, garbage out! Antiquated expression, but I like it.
Glitch	A spike of electrical noise. You don't want any of these. Cen destroy your memory contents.
Hard Copy	A peper printout of your progrem or date is celled herd copy.
Hardware	Hardwere is the physical bits and pieces (chips etc.) that meke up your computer.
Hertz (Hz.)	Measure of frequency meening cycles per second.
Hex	In everyday mathematics we use decimal, or base 10. Hexadecimal is a number system in base 16 and uses the numbers 0 to 9 and letters A to F (representing 10 through 15).
Input	Information placed into the computer's memory is input date, and may originete from, for example, the keyboard.
I/O	Abbreviation of Input/Output.
Integer	A whole number.
Interface	Software or hardware, or both, used to enable the computer and a peripheral to talk to each other.
Joystick	Used mainly to enable games to be played on a computer. We all know what e joystick is anyway, don't we?
Kilo (K)	Generally means one thousand, except when referring to memory size when it means 1024.
Line Number	The number required at the beginning of a line in BASIC is its line number. The program is always executed in line number order, unless you use something like a GOTO or GOSUB statement
LOAD	The placing of data in memory from a backing store or program.
Location	Same as absolute address.
Machine Code	Literally the language the computer understands. Machine code is the language all other languages have to be translated into before the computer can execute a program.
Memory	Storage inside the computer for data and programs, measured in bytes.
Menu	List of choices open to the user, usually encountered as the first page, or screen of e program.
Microcomputer	A small computer using a microprocessor chip. In the MTX series computers, the microprocessor is the Zilog Z80.
Microprocessor	The chip used in your computer as its CPU. Microprocessors crop up everywhere these days, in ovens, Hi Fi equipment, they are even responsible for telling you to put your seatbelt on in a Maestro.
Microsecond (us)	One millionth of a second.
Millisecond (ms)	One thousandth of a second.
Monitor	Think of it as a high definition television that can be used only as a display screen.
Nibble	Half a byte, i.e. usually four bits.
Non-Volatile	Most of the contents of memory are lost when the power is turned off. Non-volatile memory doesn't disappear. For example, the information in ROM is non-volatile.
Null String	An empty string. The string must exist, and it must have nothing in it for it to be a null string.

Number-Crunching	Performing complex calculations quickly
Object Code	A form of code the computer understands. If you write your program in a high level language, (source code) it has to be translated into object or machine code before the computer can act on it. This is a binary version of the source code and is produced by the compiler
On-Line	Peripherals connected to and communicating with a computer are said to be on-line.
Operand & Operator	Machine code instructions can be divided into these two parts. The operator is the process which is carried out, e.g. add, subtract, etc. and the operand is the data the process is carried out on, usually a number.
Operating System (OS)	Software which supervises the running of other programs. CP/M, developed by Digital Research Inc. in 1976 is an excellent operating system for use with Z80 microprocessor computers like the MTX series
Output	The results that the computer makes available to the user (either on the screen or as a printout, maybe)
Overflow	When the space allowed for the answer of an arithmetic expression is too small, an overflow condition will occur. The Z80 CPU has an overflow flag.
Pack	A way of compacting information to economise on storage space inside a computer.
Page	A block of data, as displayed by the television set or monitor. Sometimes a page is made up with several frames, or screens, of data.
Paging	Switching between blocks of computer memory
Peek	A BASIC command which allows you to read the contents of a specified memory address
Peripherals	Devices linked to the computer to enable it to gather and display information, e.g. a printer, or a TV screen are peripherals
Pixel	Picture element. It's the smallest area of display that the computer can control. The more pixels you've got, the higher the resolution of your computer.
Poke	BASIC command which places integer values into a specified memory location.
Port	A socket on the computer into which an I/O device can be plugged
Program	A set of instructions which the computer carries out.
PROM	Programmable Read Only Memory
RAM	Random Access Memory.
Record	A grouped set of related data or information. A file is generally made up of lots of records.
Register	A special storage location in the CPU which holds data on which calculations are performed
Reset	On the MTX series computers the two keys on either side of the space bar are the Reset Keys. Reset means the same as Initialise, and once pressed, the computer returns to the state it was in when you first switched on.
Reserved Word	A word that has a specific meaning to the compiler, so it cannot be used as a variable name in a program
ROM	Permanent Read Only Memory
RS232	A type of interface
Run	A command used to tell the computer to execute a program.
Scroll	The continuous movement of the display on the screen. Usually scrolling means that the latest line entered is added at the bottom and all the other lines move up one, causing the top line to disappear from view
Software	The program itself i.e. as opposed to Hardware
Source Code	What is actually written by the programmer before it is converted to object or machine code.
String	A sequence of records, words, letters or numbers
Subroutine	Often a part of a program will need to be repeated several times during the 'run'. Instead of writing the section each time you need it, a subroutine means you can write it just once, and 'call' or use it as needed
Syntax	Computer languages are very precise. Statements need to be in the correct order in the program, or it will crash. The rules which decide the grammar of the language are its Syntax
Variable	An element of a program that can have various values. It is a label used to refer to an area of memory
Volatile	Opposite of non-volatile
Zilog	'The last word in integrated logic'. The manufacturer of the Z80 micro chip used in the MTX series computers

MTX SERIES SOFTWARE APPENDICES

APPENDIX 1 : CODES OF CONTROL

All of the following control sequences are achieved by pressing the CTRL key and its parameter simultaneously.

CTRL A	PLOT X,Y
CTRL B	LINE X1,Y1,X2,Y2
CTRL C	CURSOR X,Y
CTL Dn	Sets background colour to n
CTL E	Erase to end of line
CTL Fn	Sets foreground colour to n
CTL G	Sounds the bell
CTL H	Backspace, cursor left
CTL I	Tabulate the next block of eight columns
CTL J	Line feed, cursor down
CTL K	Cursor up
CTL L	Clear screen and home cursor
CTL M	Carriage return, cursor to left edge of screen
CTRL N	CTLSPR P,X
CTRL O	GENPAT P,N,D1,D2,D3,D4,D5,D6,D7,D8
CTRL P	COLOUR P,N
CTRL Q	ADJSPR P,N,V
CTRL R	SPRITE N,P,XP,YP,XS,YS,COL
CTRL S	MOVSPR P,N,D
CTRL T	VIEW DIR,DIS
CTRL U	INSERT KEY
CTRL V	DELETE KEY
CTRL W	BACK TAB
CTRL Y	TAB KEY
CTRL Z	HOME KEY
CTL]	Page mode
CTL \	Scroll mode
CTL ^	Cursor on
CTL _	Cursor off

APPENDIX 2 : ESCAPE CODES

All of the following ESCape sequences are activated by pressing the ESC key followed by the escape character(s), (do not press ESC and the characters simultaneously).

ESC 'A'	ATTR P,STATE
ESC 'Y'	CRVS N,T,X,Y,W,H,S
ESC 'Z'	VS N
ESC 'C'	GR$ X,Y,B (RESULT IN WKAREA)
ESC B0	American character font
ESC B1	English character font
ESC B2	French character font
ESC B3	German character font
ESC B4	Swedish character font
ESC B5	Spanish character font
ESC I	Inserts a blank line at cursor line
ESC J	Deletes the current cursor line
ESC K	Duplicates a line
ESC Xc	Simulates CONTROL character c

APPENDIX 3 ERROR MESSAGES

Params
Incorrect or wrong number of parameters for a function or command.
Mistake
A mistake has been made which should be obvious from the context.
A
Dot outside virtual screen.

SE.A
Screen type not in type table.

SE.B
Invalid ESC sequence.

SE.C
Command not valid for this device.

SE.D
Switch to absent Virtual Screen.

SE.E
Invalid UDG/UDG type.

Symbol?
A symbol is missing, such as
"/" "TO", "THEN",","
Not numeric
A number is expected.

Not a string
A string is expected.

Boolean?
A truth value is expected.

Mismatch
An illegal relationship between different types of values.
BK
Break in tape LOAD or SAVE.

No data
No data for READ or No page for NODDY.

Overflow
Number too big.

Div/0
Division by zero.

Out of range
Number is not in a valid range.

No space
To define an array
To expand a program
To assign a string to a character array
To perform a large operation.

Subscript
A Subscript is out of range or there are too many.

Gosub
Too many GOSUBS (more than 34).

Undefined
A variable is being used before it exists.

Array exists
An array has already been defined.

No FOR
A next has been encountered without a matching FOI

No call
A RETURN has been encountered without a matchinç GOSUB.
No line
A reference is made to a non-existent line.

APPENDIX 4 : ASCII CODES

ASCII	HEX	DEC	ASCII	HEX	DEC	ASCII	HEX	DEC
NUL	00	0	/	2F	47	■	5E	94
SOH	01	1	0	30	48	—	5F	95
STX	02	2	1	31	49	■	60	96
ETX	03	3	2	32	50	a	61	97
EOT	04	4	3	33	51	b	62	98
ENQ	05	5	4	34	52	c	63	99
ACK	06	6	5	35	53	d	64	100
BEL	07	7	6	36	54	e	65	101
BS	08	8	7	37	55	f	66	102
HT	09	9	8	38	56	g	67	103
LF	0A	10	9	39	57	h	68	104
VT	0B	11	:	3A	58	i	69	105
FF	0C	12	;	3B	59	j	6A	106
CR	0D	13	<	3C	60	k	6B	107
SO	0E	14	=	3D	61	l	6C	108
SI	0F	15	>	3E	62	m	6D	109
DLE	10	16	?	3F	63	n	6E	110
DC1	11	17	■	40	64	o	6F	111
DC2	12	18	A	41	65	p	70	112
DC3	13	19	B	42	66	q	71	113
DC4	14	20	C	43	67	r	72	114
NAK	15	21	D	44	68	s	73	115
SYN	16	22	E	45	69	t	74	116
ETB	17	23	F	46	70	u	75	117
CAN	18	24	G	47	71	v	76	118
EM	19	25	H	48	72	w	77	119
SUB	1A	26	I	49	73	x	78	120
ESC	1B	27	J	4A	74	y	79	121
FS	1C	28	K	4B	75	z	7A	122
GS	1D	29	L	4C	76	■	7B	123
RS	1E	30	M	4D	77	■	7C	124
US	1F	31	N	4E	78	■	7D	125
space	20	32	O	4F	79	■	7E	126
!	21	33	P	50	80	DEL	7F	127
"	22	34	Q	51	81			
■	23	35	R	52	82			
■	24	36	S	53	83			
%	25	37	T	54	84			
&	26	38	U	55	85			
'	27	39	V	56	86			
(	28	40	W	57	87			
)	29	41	X	58	88			
*	2A	42	Y	59	89			
+	2B	43	Z	5A	90			
,	2C	44	■	5B	91			
-	2D	45	■	5C	92			
.	2E	46	■	5D	93			

■ Denotes characters that change according to which character set is selected. See table

Hex. code \ Country	U.S.A.	France	Germany	England	Denmark	Sweden	Italy	Spain
23	#	#	#	£	#	#	#	Pt
24	$	$	$	$	$	¤	$	$
40	@	à	§	@	@	É	@	@
5B	[	°	Ä	[	Æ	Ä	°	¡
5C	\	ç	Ö	\	Ø	Ö	\	Ñ
5D	]	§	Ü	]	Å	Å	é	¿
5E	^	^	^	^	^	Ü	^	^
60	'	'	'	'	'	é	ù	'
7B	{	é	ä	{	æ	ä	à	¨
7C	¦	ù	ö	¦	ø	ö	ò	ñ
7D	}	è	ü	}	å	å	è	}
7E	~	¨	ß	~	~	ü	ì	~

APPENDIX 5 : CODES OF COLOUR

The following table provides a list of the colours and the numbers by which the Memotech will recognise them.

0 Transparent
1 Black
2 Medium Green
3 Light Green
4 Dark Blue
5 Light Blue
6 Dark Red
7 Cyan
8 Medium Red
9 Light Red
10 Dark Yellow
11 Light Yellow
12 Dark Green
13 Magenta
14 Grey
15 White

APPENDIX 6 : KEYWORD ABBREVIATIONS

ABS	AB.	GOTO	G.	PLOT	-
ADJSPR	AD.	GR$	GR.	POKE	PO.
AND	-	IF	-	PRINT	P.
ANGLE	ANG.	INK	I.	RAND	RA.
ARC	AR.	INKEY$	INKE.	READ	REA.
ASC	-	INP	-	REM	R.
ASSEM	A.	INPUT	INP.	RESTORE	RES.
ATN	-	INT	-	RETURN	RET.
ATTR	AT.	LEFT$	LEF.	RIGHT$	RIG.
AUTO	AU.	LEN	-	RND	RN.
BAUD	B.	LET	LE.	RUN	RU.
CHR$	CH.	LINE	LIN.	SAVE	SA.
CIRCLE	CI.	LIST	L.	SBUF	SB.
CLEAR	CLE.	LLIST	LL.	SGN	SG.
CLOCK	CLO.	LN	-	SIN	SI.
CLS	C.	LOAD	LO.	SOUND	SO.
COLOUR	COL.	LPRINT	LP.	SPK$	SPK.
CONT	CO.	MID$	MI.	SPRITE	S.
COS	-	MOD	MO.	SQR	SQ.
CRVS	CR.	MVSPR	MV.	STEP	STE.
CSR	CS.	NEW	-	STOP	STO.
CTLSPR	CT.	NEXT	N.	STR$	STR.
DATA	D.	NODDY	NODD.	TAN	TA.
DB		NODE	NOD.	THEN	T.
DIM	DI.	NOT	-	TIME$	TI.
DRAW	DR.	ON	O.	TO	-
DS		OR	-	USR	-
DSI	DS.	OUT	OU.	USER	U.
EDIT	E.	PANEL	PAN.	VAL	VA.
EDITOR	EDITO.	PAPER	PA.	VERIFY	VE.
ELSE	EL.	PAUSE	PAU.	VIEW	VI.
EXP	EX.	PEEK	-	VS	V.
FOR	F.	PHI	PH.		
GENPAT	GE.	PI	-		
GOSUB	GOS.	PLOD	PL.		

APPENDIX 7 : DEGREE TO RADIAN CONVERSION CHART

DEGREES	RADIANS
360	2*PI
270	3*PI/2
180	PI
90	PI/2
60	PI/3
45	PI/4
30	PI/6
22.5	PI/8
15	PI/12
10	PI/18
7.5	PI/24
5	PI/36
1	PI/180

APPENDIX 8 : FUNCTION KEYS

The Function Keypad can be used to customise the computer for a particular application. There are eight keys marked F1 to F8.

Try this program:

```
10 PRINT ASC(INKEY$)
20 GOTO 10
```

If you press any key, you will see its ASCII code displayed and the shifted value if the shift key is pressed simultaneously.

F1	128	SHIFT	and	F1	136
F2	129	"	"	F2	137
F3	130	"	"	F3	138
F4	131	"	"	F4	139
F5	132	"	"	F5	140
F6	133	"	"	F6	141
F7	134	"	"	F7	142
F8	135	"	"	F8	143

If required, character patterns can be assigned to the function keys using the GENPAT statement. For example,

```
10 GENPAT 1,129,32,80,136,136,248,136,136,0
```

will make F2 produce a character 'A'.

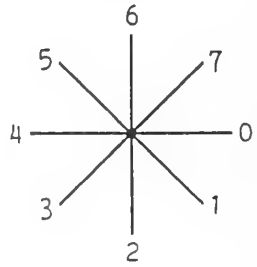
THE NUMERIC KEYPAD

The Numeric Keypad has been designed for use with application programs. Notice that the Break key is in the top right hand corner of the numeric pad and you must decide if you want to allow this key to Break in or not. To use the numeric pad, press the shift key and one of the numbers. In this mode the Break/9 key will give a 9 and not Break.
There is however a numeric padlock which is set by a Bit in the keyboard flags. If this bit is set, the number pad will be locked to Numbers but for safety the Break key will over-ride the 9. To use the 9 you must turn off the Break key, by switching the Break key bit in the Interrupt flags INTFFF.

```
e.g.
10 POKE 64145,132
20 POKE 64862,13
30 PRINT INKEY$:GOTO 30
```

APPENDIX 9 : ABSOLUTE DIRECTIONS

Some graphics commands, including MVSPR, and VIEW, use a direction parameter to specify one of seven directions. These are illustrated in the diagram below.



APPENDIX 10 : SYSTEM VARIABLES

FA52	CTRBADR	DS 40	Control buffers for sound
FA7A	LSTPG	DS 1	This contains the number of 32K RAM pages present - 1
FA7B	VARNAM	DS 2	This contains the address of the bottom of the variable name table
FA7D	VALBOT	DS 2	This contains #FF. VALBOT plus 1 is the address of the bottom of the variable value table
FA7F	CALCBOT	DS 2	This contains the address of the bottom of the calculator stack
FA81	CALCST	DS 2	Stack Pointer This contains the address of the top of the calculator stack + 1, ie the next available free byte
FA83	KBDBUF	DS 2	This contains the address of the Keyboard Buffer

FA85 USYNT DS 4

This contains the syntax bytes which are used to tell the computer what to expect when the BASIC command word USER is met. These bytes may be defined by the operator, as listed below. They are examined from the top of the four byte block to the bottom, and the last one must contain a RET instruction.

FA89 USER DS 3

This contains the address of the routine which will be jumped to when the BASIC command word USER is met. It usually contains RET, but may be redefined. If you wish to put a new jump address into USER, it is important that it is changed in reverse order, ie #FA8B first, then #FA8A, then #FA89, otherwise the computer will jump to 0000, which is equivalent to a RESET. This basic idea applies to all jump locations.

eg original contents of USER.....

```
FA89 C9     RET
FA8A 00     not used
FA8B 00     yet

to force a jump to #D0FF

FA89 C3     JUMP instruction to....      <change last
FA8A FF     Low byte first, then         <change second
FA8B D0     High byte                    <change first
```

Effect of various Syntax Bytes in the USYNT location

Syntax byte (decimal)	Syntax checked for
0	Numeric Expression
1	String Expression
2	Arithmetic Expression
3	List of Expressions separated by "," or ";"
4	List of numbers separated by "," in range 0 to 64K
5	List of Arithmetic Expressions
6	Single number in range 0 to 64K
7	Allows anything, ie no checking
8	Checks syntax for INPUT statement
9	Checks syntax for IF statement
10	Checks syntax for STEP in FOR statement
11	GOTO or GOSUB
12	I=(arithmetic expression) in FOR statement
13	Numeric variable or nothing

Any value greater than 32 will cause the computer to expect that value to be input.

FA8C		DS 3	Not used by computer, however a JUMP address here will be saved to tape.
FA8F	IOPL	DS 1	Pointer to the List device; see below.
FA90	REALBY	DS 1	PANEL breakpoint Stores byte of breakpoint in GO,
FA91	KBFLAG	DS 1	See below.
FA92	STKLIM	DS 2	STACK Limit. Points to top of free space.
FA94	SYSTOP	DS 2	Points to top of variables to be saved.
FA96	SSTACK	DS 2	Points to start of Machine Stack. This value is loaded into the SP (Stack Pointer) register whenever the machine goes back to basic, or an error occurs.
FA98	USERINT	DS 3	Contains JUMP address used dependent upon the bit set up in INTFFF at #FD5E; see below.
FA9B	NODLOC	DS 3	Contains JUMP address used by the MTX RING Local Area Network.
FA9E	FEXPAND	DS 3	PANEL expansion. Before executing a PANEL command the computer looks here for a JUMP instruction. Normally contains RET.

FAA1	USERNOD	DS 3	NODDY expansion. Works as FEXPAND above.
FAA4	NBTOP	DS 2	Top of NODDY in current page
FAA6	NBTPG	DS 1	Current NODDY page.
FAA7	BASTOP	DS 2	Top of BASIC in current page
FAA9	BASTPG	DS 1	Current BASIC page
FAAA	BASBOT	DS 2	Value from which virtual adresses are calculated, normally #4000
FAAC	BASTPO	DS 32	Top of each BASIC page, (max 16)
FACC	ARRTOP	DS 3	Points to top of Arrays
FACF	BASELIN	DS 2	Contains number of current line being executed.
FAD1	BASLNP	DS 1	Contains page number for BASELIN
FAD2	PAGE	DS 1	Contains current page configuration.
FAD3	CRNTPG	DS 1	Contains number of current BASIC page.
FAD4	PGN1	DS 1	Space for temporary variable used by BASIC Interpreter
FAD5	PGN2	DS 1	Space for temporary variable used by BASIC Interpreter
FAD6	PGTOP	DS 2	Contains address of top of current page
FAD8	GOSTACK	DS 105	GOSUB Stack. Stores GOSUB return addresses
FB41	GOPTR	DS 2	Pointer into GOSUB stack for next return address
FB43	GOSNUM	DS 1	Number of GOSUBS on GOSUB stack
FB44	FORCOUNT	DS 1	Counter of No. of nested FOR loops
FB45	CTYSLT	DS 1	Selects Keyboard configuration for different countries
FB46	DATAAD	DS 2	Data pointer for READ statements
FB48	DATAPG	DS 1	Page number holding data above
FB49	DESAVE	DS 2	Stores the current BASIC program position on saving to tape.

----------System variables saved to here----------

FB4B	START	DS 200H-3	Start of Keyboard Buffer
FD48	SETCALL	DS 3	Temporary jump address location used for Single stepping in PANEL
FD4B	RICHJL	DS 3	Temporary variable used for Single stepping in PANEL
FD4E	USRRST	DS 3	This location is examined on calling RST 38 or on a Non Maskable Interrupt. It normally contains RET.
FD51	USERIO	DS 3	This location is examined on calling KBD at #79. KBD reads the Keyboard, and leaves the result in A.
FD54	USERROR	DS 3	This location is examined before an error message is displayed.
FD57	CLOCK	DS 7	Real Time Clock. Contains time in format: Byte no:1 2 3 4 5 6 7 H H M M S S X where X counts up from 48 to 173 in 125ths of a second. The data is displayed in PANEL as ASCII.
FD5E	INTFFF	DS 1	See below
FD5F	CASBAUD	DS 1	Cassette BAUD rate
FD60	MIDVAL	DS 1	Contains a reference value for tape save, load and verify, and varies with different BAUD rates. Eg if the length of a "1" pulse is 100 units and the length of a "0" pulse is 50 units, then MIDVAL will be approx 75 units for that BAUD rate.
FD61	RETSAVE	DS 4	Start address for auto load
FD65	VAZERO	DS 2	Virtual Address of bottom of BASIC
FD67	VERIF	DS 1	Flag for VERIFY or LOAD
FD68	TYPE	DS 1	Flag for SAVE or LOAD

FD69	CONTFLG	DS 1		Continue flag. 0 implies cannot continue after BREAK key pressed or STOP command
FD6A	CONTAD	DS 2		Address of BASIC line to continue from after BREAK key or STOP
FD6C	CONTPG	DS 1		Page number for above
FD6D	ASTACK	DS 2		Contains address of Machine stack used by PANEL
FD6F	TMPHL	DS 2		Stores HL during page switching
FD71	TMPA	DS 2		Stores A during page switching
FD73	STACCT	DS 2		Temporary variable used by maths routines
FD75	PRORPL	DS 1		See below
FD76	IOPR	DS 1		See below
FD77	AUTOIN	DS 2		Increment for Auto Line
FD79	AUTOST	DS 2		Start value for Auto Line
FD7B	AUTOCT	DS 1		Counter for Auto Repeat
FD7C	LASTKY	DS 1		Last key pressed
FD7D	LASTASC	DS 1		ASCII of last key read
FD7E	LASTDR	DS 1		Current drive line in use in keyboard scan
FD7F	RNSEED	DS 2		Seed for random number routine
FD81	BREAK	DS 1		Break key flags
FD82	COMMAND	DS 2		Contains address of first command executed if direct
FD84	ERRPOS	DS 2		Contains address of syntax error
FD86	FLAGS1	DS 1		Flags displayed in PANEL
FD87	ITYPE	DS 2		Temporary variable used by PANEL
FD89	MAFD	DS 2	*	
FD8B	MBCD	DS 2	*	Used by PANEL for temporarily storing alternate register set
FD8D	MDED	DS 2	*	
FD8F	MHLD	DS 2	*	

```
FD91            MAF         DS 2   *
FD93            MBC         DS 2   *
FD95            MDE         DS 2   *
FD97            MHL         DS 2   *    Used by PANEL for temporarily
FD99            MIX         DS 2   *    storing register set
FD9B            MIY         DS 2   *
FD9D            MSP         DS 2   *
FD9F            MPC         DS 2   *

FDA1            MEMPOINT    DS 2        Pointer into memory for PANEL
                                        display


FDA3            WCHJUMP     DS 2        Variables used by Assembler
FDA5            POINTERR    DS 1        and PANEL.........
FDA6            DADD        DS 2
FDA8            INDEX       DS 2
FDAA            DBYTE       DS 2
FDAC            LINKER      DS 1
FDAD            EDIT        DS 1
FDAE            LENGTH      DS 1
FDAF            DETYPE      DS 1
FDB0            DTYPE       DS 1
FDB1            DISAD       DS 2
FDB3            DPROG       DS 2
FDB5            LABTABL     DS 2        Symbol table address
FDB7            APROG       DS 2        Start of assembly code program
FDB9            ENDTAB      DS 2
FDBB            COMMENT     DS 1
FDBC            COMAD       DS 2
FDBE            ADLABEL     DS 3
FDC1            INDEXLAB    DS 2
FDC3            DATALAB     DS 3
FDC6            DBLABEL     DS 1
FDC7            BASEM       DS 2
FDC9            CURLAB      DS 2
FDCB                        DS 1

FDCC            ACC1        DS 5        Accumulator used by Maths routines,
                                        contains five byte floating point
                                        number

FDD1            OP1         DS 1 *
FDD2            OP11        DS 5 *      Temporary variables used by maths
FDD7            YORN        DS 1 *
FDD8            SIGN        DS 1 *
FDD9            MEM1        DS 5 *      Temporary location for a
                                        floating point number

FDDE            COPY        DS 20       Contains copy of tape header
                                        information

FDF2            INTTAB      DS 16       Interface for European Keyboard
                                        layouts

FE02            GASH        DS 2        Temporary variable used by sound

FE04            TEMP        DS 16       Temporary variable used by sound
```

FE14		CHAN	DS 2	These three locations hold the three parameters used by the direct sound statement. They can be poked with the ranges of values used by this command. Calling #8F6 will set the sound chip going using these values
FE16		FREQ	DS 2	
FE18		VOL	DS 2	
FE1A		WKAREA:	DS 37	Escape sequence data collection area
FE3F	00 00 00 00	BSSTR:	DEFS 12	Screen Workarea
FE43	00 00 00 00			
FE47	00 00 00 00			
FE4B	01	SPEED:	DEFB 1	Unit of sprite speed
FE4C	01	SPBASE:	DEFB 1	Temporary variable
FE4D	01	MVDIST:	DEFB 1	Unit of distance for MVSPR
FE4E	00	NOSPR:	DEFB 0	Number of active sprites
FE4F	00	DLSPNO:	DEFB 0	Number of circling sprites
FE50	00	PLSPNO:	DEFB 0	Current Plot Sprite number
FE51	00	MVNO:	DEFB 0	Number of sprite for move command
FE52	00	DELSPR:	DEFB 0	Temporary variable
FE53	00	VCOUNT:	DEFB 0	Counter for cursor flash
FE54	00	VDPSTS:	DEFB 0	Copy of VDP status register
FE55		SPRTBL:	DS 256	Control buffers for sprites
FF55	00	SMBYTE:	DEFB 0	Size and magnitude of sprites
FF56	00	LENLO:	DEFB 0	Control variable for ARC
FF57	00	LENHI:	DEFB 0	Control variable for ARC
FF58	00	VINTFG:	DEFB 0	Sprite interrupt flag: if this contains zero it is safe to write to the screen
FF59	00 00	CHPTR:	DEFB 0,0	Character pointer
FF5B	00 00	CURSCR:	DEFB 0,0	Points at start of current screen data below

These blocks of data contain the virtual screen parameters used at switch-on. They may be changed using CRVS, POKE or from Assembler code - see Virtual screen byte format table below

```
FF5D    00 00 00 00       SCRN0:      DEFB 0,0,0,0,0,40,24,40,0,241
FF61    00 28 18 28
FF65    00 F1
FF67    F1 00 F1 00                   DEFB 241,0,241,0,0
FF6B    00

FF6C    00 00 00 00       SCRN1:      DEFB 0,0,0,0,0,28,24,40,0,241
FF70    00 1C 18 28
FF74    00 F1
FF76    F1 00 F1 00                   DEFB 241,0,241,0,0
FF7A    00

FF7B    00 00 00 1C       SCRN2:      DEFB 0,0,0,28,0,12,24,40,0,241
FF7F    00 0C 18 28
FF83    00 F1
FF85    F1 00 F1 00                   DEFB 241,0,241,0,0
FF89    00

FF8A    00 00 00 0A       SCRN3:      DEFB 0,0,0,10,5,20,14,40,0,31
FF8E    05 14 0E 28
FF92    00 1F
FF94    1F 00 1F 00                   DEFB 31,0,31,0,0
FF98    00


FF99    20 00 00 00       SCRN4:      DEFB 32,0,0,0,0,32,24,32,0,241
FF9D    00 20 18 20
FFA1    00 F1
FFA3    F1 00 F1 00                   DEFB 241,0,241,0,0
FFA7    00

FFA8    20 00 00 00       SCRN5:      DEFB 32,0,0,0,16,12,8,32,0,241
FFAC    10 0C 08 20
FFB0    00 F1
FFB2    D1 00 1F 00                   DEFB 209,0,31,0,0
FFB6    00

FFB7    20 00 00 00       SCRN6:      DEFB 32,0,0,0,16,12,8,32,0,31
FFBB    10 0C 08 20
FFBF    00 1F
FFC1    D1 00 1F 10                   DEFB 209,0,31,16,0
FFC5    00

FFC6    00 00 00 00       SCRN7:      DEFB 0,0,0,0,0,0,0,0,0,0
FFCA    00 00 00 00
FFCE    00 00
FFD0    00 00 00 00                   DEFB 0,0,0,0,0
FFD4    00
```

FFD5 TYPTBL: DS 24

This area holds the page number and address of the routines for particular screen types. The table is organised into three byte blocks, allowing up to 8 screen types. Currently only four screen types are defined:

Screen Type	Characteristics
0	40 column text only
1	32 column graphics
2	40 column text and graphics
3	80 column text and graphics
4 - 7	Undefined

FFED 00 OVRLAY:

This is a 3 byte JUMP location which is examined prior to accessing a virtual screen. It allows the user to redefine the format of the next screen to be called.

FFF0 IJTABLE DS 16 Interrupt jump table

Virtual Screens - byte format for each screen

Byte(s)	Contents
1	Various Bits indicate Screen type, Auto Scroll, Cursor flash, Page mode
2+3	Current PRINT position within this screen in col,row format, eg could be #0F #0F indicating 15,15
4+5	Absolute top left hand corner, represented as above
6+7	Width of screen, and no. of lines in chars
8	Line width of physical screen
9	Contains the character under the cursor
10	Shows Border colour
11	Shows PRINT colours as INK, PAPER
12	Shows PRINT attributes
13	Shows PLOT colours as INK, PAPER
14	Shows PLOT attributes
15	Counter for number of lines scrolled so far

```
INTFFF          FD5E
USERINT         FA98
```

Every 125th of a second an interrupt occurs, the clock is updated, and bit 7 of INTFFF is toggled. If bit 7 of INTFFF is 1, and any of the USER bits in INTFFF are 1 a call is made to the USERINT location. If all three USER bits in INTTFFF are set, USERINT is called three times.

```
INTFFF Bit 0    Sound routine called
           1    Break key tested
           2    Keyboard autorepeat enabled
           3    Sprite movement and cursor flash enabled
           4    USER bit-USERINT called if set
           5    USER bit-USERINT called if set
           6    USER bit-USERINT called if set
           7    Clock bit-toggled every 125th of a second

LASTDR          FD7E
```

During an interrupt the keyboard is scanned to see if the break key has been pressed. Because this involves outputting data on the keyboard drive lines, the interrupt routine may upset a user initiated keyboard scan. To prevent a conflict the operator should load LASTDR with the data to be output on the keyboard drive line immediately before performing the output. The interrupt routine outputs this data again before returning to guarantee that the drive lines are left the same as when the interrupt routine was entered.

```
PRORPL          FD75
IOPL            FA8F
IOPR            FD76
```

If PRORPL = 1, output will be sent to the device specified by IOPL
If PRORPL = 0, output will be sent to the device specified by IOPR

IOPR and IOPL can be set up as follows:

IOPR/IOPL = 0 output goes to the screen
IOPR/IOPL = 1 output goes to the Centronics printer port
IOPR/IOPL = 2 output goes to the RS232 port channel 0

```
KBDFLG          FA91
```

Bits set have the following effect:

```
Bit  7  set  Alpha lock on
     5  set  Toggles between page and scroll modes
     2  set  Locks numeric keypad
```

APPENDIX 11 : SOUND TABLE 1

MTX/HERTZ FREQUENCIES

FREQUENCY = 4000000/32*n (where n is the value)
Direct sound frequency parameter=125000/Hz frequency
Continuous sound frequency parameter=1000000/Hz frequency

* DIRECT	** SBUF	RESULT (Hz)
10	80	12500
20	160	6250
30	240	4166
40	320	3125
50	400	2500
60	480	2083
70	560	1785
80	640	1562
90	720	1388
100	800	1250
110	880	1136
120	960	1041
130	1040	961
140	1120	892
150	1200	833
160	1280	781
170	1360	735
180	1440	694
190	1520	657
200	1600	625
210	1680	595
220	1760	568
230	1840	543
240	1920	520
250	2000	500
260	2080	480
270	2160	462
280	2240	446
290	2320	431
300	2400	416
310	2480	403
320	2560	390
330	2640	378
340	2720	367
350	2800	357
360	2880	347
370	2960	337
380	3040	328
390	3120	320
400	3200	312
420	3360	297
440	3520	284
460	3680	271
480	3840	260
500	4000	250
520	4160	240
540	4320	231

DIRECT	SBUF	RESULT (Hz)
560	4480	223
580	4640	215
600	4800	208
620	4960	201
640	5120	195
660	5280	189
680	5440	183
700	5600	178
720	5760	173
740	5920	168
760	6080	164
780	6240	160
800	6400	156
820	6560	152
840	6720	148
860	6880	145
880	7040	142
900	7200	138
920	7360	135
940	7520	132
960	7680	130
980	7840	127
1000	8000	125
1020	8160	122

*(for three parameter SOUND command)

**(for seven parameter SOUND command)

APPENDIX 12 : SOUND TABLE 2

FREQUENCIES FOR CHROMATIC SCALE OF C

The table below lists the frequency values for the chromatic scale of middle C.

NOTE	HERTZ	DIRECT	CONTINUOUS
C	256	488	3906
C#	271	460	3687
D	287	435	3480
D#	304	411	3285
E	322	388	3100
F	342	366	2926
F#	362	345	2762
G	384	326	2607
G#	406	308	2461
A	430	290	2323
A#	456	271	2192
B	483	259	2069
C	512	244	1953

APPENDIX 13 : SOUND TABLE 3

NOISE

DC (periodic noise)	SBUF	R
0	0	Shift rate = 7812.5 Hz
1	8	Shift rate = 3906.25 Hz
2	16	Shift rate = 2604.17 Hz
3	24	Shift rate = CHANNEL 2
Pink Noise		
4	32	Shift rate = 7812.5 Hz
5	40	Shift rate = 3906.25 Hz
6	48	Shift rate = 2604.17 Hz
7	56	Shift rate = CHANNEL 2

APPENDIX 14 : SOUND TABLE 4

VOLUME

Direct Com.	SBUF	Result (DB)
0	0	OFF
1	256	– 0
2	512	– 2
3	768	– 4
4	1024	– 6
5	1280	– 8
6	1536	–10
7	1792	–12
8	2048	–14
9	2304	–16
10	2560	–18
11	2816	–20
12	3072	–22
13	3328	–24
14	3584	–26
15	3840	–28

MTX SERIES TECHNICAL APPENDICES

Some of the illustrations used in the technical appendix appear by permission of Zilog Inc and Texas Instruments Inc.

1 INTRODUCTION

Overall Description

The MTX500 Series personal computer systems are high performance 8-bit computers uniquely designed to operate in memory intensive ROM-based or DISC-based environments. The choice of the Z80A Microprocessor and the TMS 9918A series video processor as the key components of the hardware architecture is consistent with a low cost ROM-based system with colour TV output plus the capability to expand to accomodate a fully RAM-based Disc operating system such as CP/M, utilising a high quality 80 column colour monitor output.

The memory size can be either 32K or 64K Bytes as standard, expandable to 512K Bytes. There is a separate 16K Byte dedicated video memory. A 24K Byte ROM contains MTX–BASIC, the system monitor, supplementary languages and utilities. The standard interfaces included are tape cassette (Read/Write to 2400 baud), Keyboard, Cartridge Port, Twin Joysticks, Parallel Centronics type printer port, uncommitted Parallel Input/Output port, Colour TV output with sound, composite video output–monochrome or colour, and audio output. Optional interfaces include a completely independent twin RS232C with buffered bus extension, Local Area Network, Colour 80 Column Board, Floppy Disc System, Silicon disc fast access RAM boards, and a Winchester Disc System.

The Keyboard consists of 79 full travel typewriter style keys mounted on a steel base plate which is fitted to the Aluminium enclosure. Aluminium was chosen for good heat dissipation, durability and RFI shielding.

2 TECHNICAL SPECIFICATION

HARDWARE

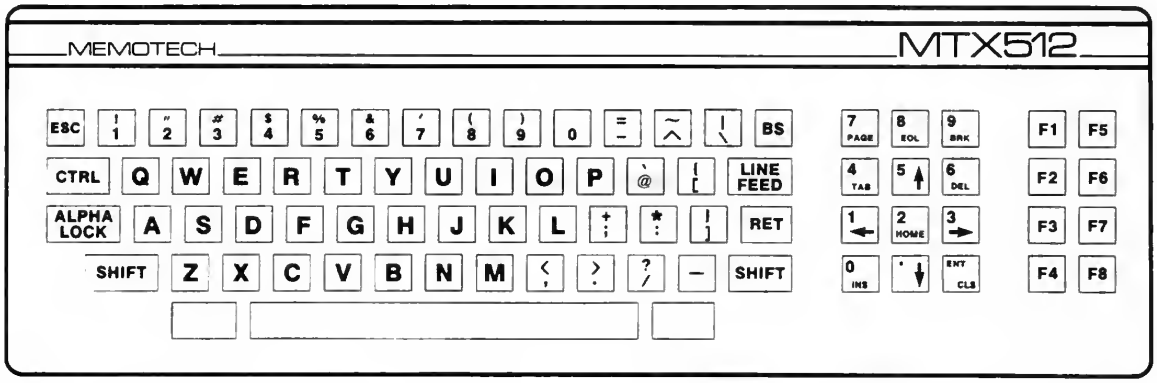
CHASSIS

Two front-hinged black anodised brushed aluminium extrusions are separated at the rear by a black plastic moulding. The extrusions act as heatsinks for the voltage regulation circuitry. Two matt black powder coated stamped aluminium endplates, are secured by 3 screws each.

Dimensions in millimetres: Width 488 Depth 202 Height 56
Weight: 2.6 kilograms

KEYBOARD

A 1mm mild steel sheet is bolted to the upper chassis and supports 79 keys which are interconnected by an independent p.c.b. The keys are arranged as:
Standard U.K. QWERTY layout with 57 professional typewriter keys, standard pitch and spacing. Keys F and J are recessed for easy fingertip location wherever possible. Foreign language keyboards are available.



Twelve dual function keys are arranged as a separate numeric keypad with cursor control and editing keys.
Eight function keys (programmable in conjunction with shift to provide 16 user definable functions).
Two unmarked keys, which must be depressed simultaneously to reset the computer.

Auto repeat is standard on all keys.

CPU BOARD

Mounted in the lower chassis, the CPU board accommodates:
Zilog Z80A CPU operating at 4MHz.
24K of ROM which contains:
MTX BASIC–incorporating sophisticated MTX LOGO-type graphics commands.
MTX NODDY–interactive text manipulation routines.
FRONT PANEL DISPLAY–incorporating Z80 Assembler/Disassembler plus Z80 Register, Memory and Program display and manipulation routines.
VIDEO DISPLAY PROCESSOR–with 16K dedicated video-RAM.
USER-RAM–32K on the MTX500 and 64K on the MTX512. User RAM size is constant under all display formats.
VIDEO BOARD–for television and sound signal encoding.

REAL TIME CLOCK

CHARACTER SETS–Numeric, upper case, lower case, user-definable characters and user-definable sprites. Resident international character sets and appropriate keyboard layouts for UK, USA, France, Germany, Spain and Sweden. Character sets for Denmark and Italy are also available.

EXPANSIONS

Up to two expansion boards may be added internally. These may be Memory (RAM) Boards, Program (ROM) Boards or the Communications Board.

MEMORY BOARDS

RAM may be increased by the addition of boards which provide 64K, 128K or 256K of memory, up to a maximum of 512K.

COMMUNICATIONS BOARD

Available as an internal expansion, this board carries two completely independent RS232 interfaces (running at up to 19 200 baud) with full handshaking and modem communication lines, and also the disc drive bus. The Communications Board is required to run the FDX disc based systems and the MTX Node/Ring System.

NODE/RING SYSTEM

Communications software and interfacing enabling construction of MTX Ring Systems. The system is interrupt driven and runs in conjunction with the twin RS232 Communications Board.

Compatibility of the memory boards and communications Board is given below.

Compatibility table of internal expansion boards

RAM boards	64K	128K	256K	Comms. board
64K	*	*	*	*
128K	*	*	*	*
256K	*	*	*	*
Comms. board	*	*	*	

* = compatible

ROM Expansions

Via the cartridge port or internal bus these provide:

Hisoft Pascal

Local Area Network software
MTX Newword, Business, Education and Games software

Display

Colour TV and/or Video Monitor
40 column x 24 line display as standard, with optional Colour 80 column board. (FDX or HDX disc based system required).
Display Facilities:
FULL SCREEN HANDLING
EIGHT USER DEFINABLE VIRTUAL SCREENS
SCREEN FORMATS
Text: 40 x 24 characters
Text with graphics: 32 x 24 text with 256 x 192 pixels in 16 colours

Graphics Facilities

Up to 32 independently controllable user definable sprites, plus pattern plane and backdrop plane. High level sprite-orientated graphics commands.

Input/Output

Provided as standard:
CASSETTE PORT (variable rate, default to 2 400 baud)
UNCOMMITTED PARALLEL INPUT/OUTPUT PORT
TWO JOYSTICK PORTS with industry standard pin-outs
FOUR CHANNEL SOUND UNDER SOFTWARE CONTROL–three independent voices plus pink noise output through TV speaker, or through separate Hi-Fi output
MONITOR OUTPUT–composite video signal (1V peak to peak)
CARTRIAGE PORT
PARALLEL PRINTER PORT (compatible with Centronics-type printers)
Available as an expansion:
COMMUNICATIONS BOARD WITH TWO RS232 INTERFACES and disc drive bus
SUITABLE PRINTERS – Centronics-type parallel printers and RS232 serial printers (requires Communications Board)

Power Supply Unit

Input: 220/240 VAC 50/60 Hz. or 110/115 VAC 50/60 Hz.
Output:22.5 VAC, 1A tapped at 18V and 9V.
Dimensions in millimetres: Width 92 Depth 110 Height 70
Weight: 1.0 kilogram
The PSU is double insulated and has a side mounted rocker switch which is internally illuminated when the unit is on. The mains transformer is located between two groups of four anti-vibration, noise absorbing rubber mounts. Extensive strain relief mouldings are incorporated in the PSU casing to support the input and output cables. The output cable terminates in a 240 degree, six pin DIN connector. The PSU is supplied as a sealed unit.

MTX Series Disc Based Systems

These are the:

Single Floppy Disc System
and the
FDX Floppy Disc System

Both of these systems require the Communications Board within the MTX computer and the twin disc system requires 64K of RAM. Both systems have the following features:
A 19 inch wide chassis comprising four black anodised brushed aluminium extrusions. Black powder coated end plates are each secured by six screws. The chassis contains a card cage which can accommodate:
One computer expansion board
One Colour 80 column board
Four silicon disc memory boards
One floppy disc controller board
An integral power supply which also powers the MTX computer.
Inputs can be 240/220 VAC 50/60 Hz or 110/115 VAC 50/60 Hz.
Parallel port for bus expansion
Two slots are provided on the front face for horizontally mounted five and a quarter inch disc drives.
External battery backup facilities are optionally available
A license to use the Digital Research Inc. CP/M 2.2 operating system is provided with the FDX and HDX systems, as is CP/M itself.

HDX Winchester Disc System. This configuration offers either 10 or 20 megabyte hard disc storage. If uses a Z80 H processor running at 8 MHZ.

The HDX includes a single 1 megabyte TEAC 5¼" floppy drive.

Colour 80 column board

Mounted in the FDX or HDX systems the board permits the use of colour programs requiring an 80 column screen running under CP/M 2.2, such as Colour Newword. Also available is the wide range of existing CP/M based software.

80 Column board-Input and Output

RGB monitor output with selectable positive/negative sync.
Monochrome composite video output, 1V peak to peak, negative sync.
Light pen input
Single channel sound
Screen display formats:
80 columns x 24 lines text (max)
160 x 96 graphics mode
Two alternate 96 element character sets with true lower case descenders.
4K ROM based graphics characters
Teletext compatibility
High speed glitch free screen update (average 25 000 baud)
The Colour 80 column board provides a complete emulation of a CP/M terminal via ROM software, and features:
Full cursor control
Vector plot, point plot
Powerful editing facilities with screen dump
Complete attribute control for colour and monochrome displays

Silicon Discs

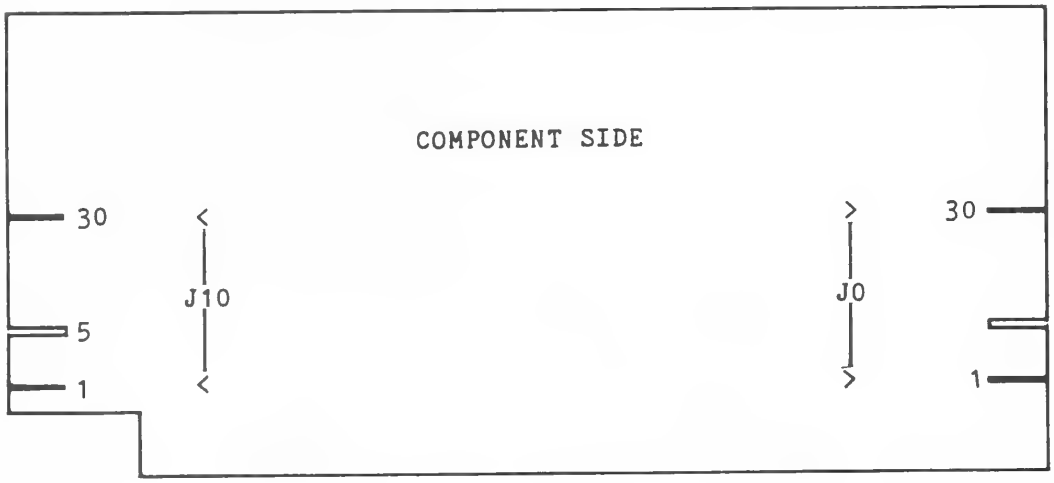
These are a quarter or one megabyte fast access RAM boards which are full emulators of CP/M drives 0 to 13. Four silicon discs may be mounted within the HDX or FDX chassis, providing from one to four megabytes per card frame. However, the silicon disc controllers can supervise four logical drives, of up to eight megabytes each giving a maximum silicon storage of 32 megabytes. This is in addition to the four five and a quarter and/or eight inch conventional floppy disc drives handled by the floppy disc controller board.

Numerous advantages include:
Speed–up to five times faster than a Winchester disc, and fifty times faster than a floppy disc.
A dramatic increase in efficiency of proven eight bit CP/M software to 16/32 bit software levels, obviating the need for complex and costly memory management techniques
Permits single floppy disc CP/M system which is ideal for database manipulation, word processing and compilation.
Greatly reduces disc wear and prolongs life of mechanical disc drives, enhancing reliability especially in disc intensive transactions.

Floppy Disc Controller Board

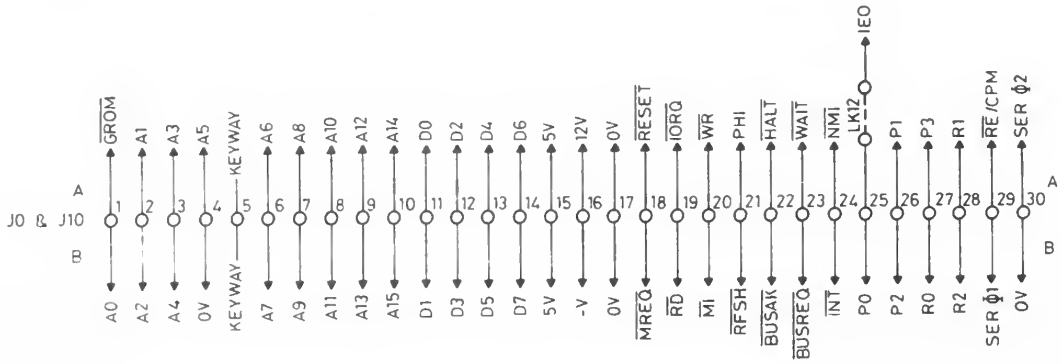
This board uses the full Western Digital 1791 chip and supports most CP/M drives, including all types 0 to 13 which range from single sided single density five and a quarter inch floppies to double sided double density eight inch floppies, using SASI (Shugart) standard interfaces. Any combination of four SASI compatible drives can be controlled. The WD 1791 controller set together with a bipolar DMA controller provides a high speed processor interface minimising latency and facilitating rapid data transfer especially on high capacity discs. Variable and fixed write precompensation is software selectable. Bus extenders permit the connection of external floppy drives.

APPENDIX 3 MTX SERIES SYSTEM BUS



The system Bus comprises the full Z80A bus, power supply rails, ROMpak enable (GROM), ROM page ports R0 to R2, RAM page ports P0 to P3 and serial clock lines 01 and 02.

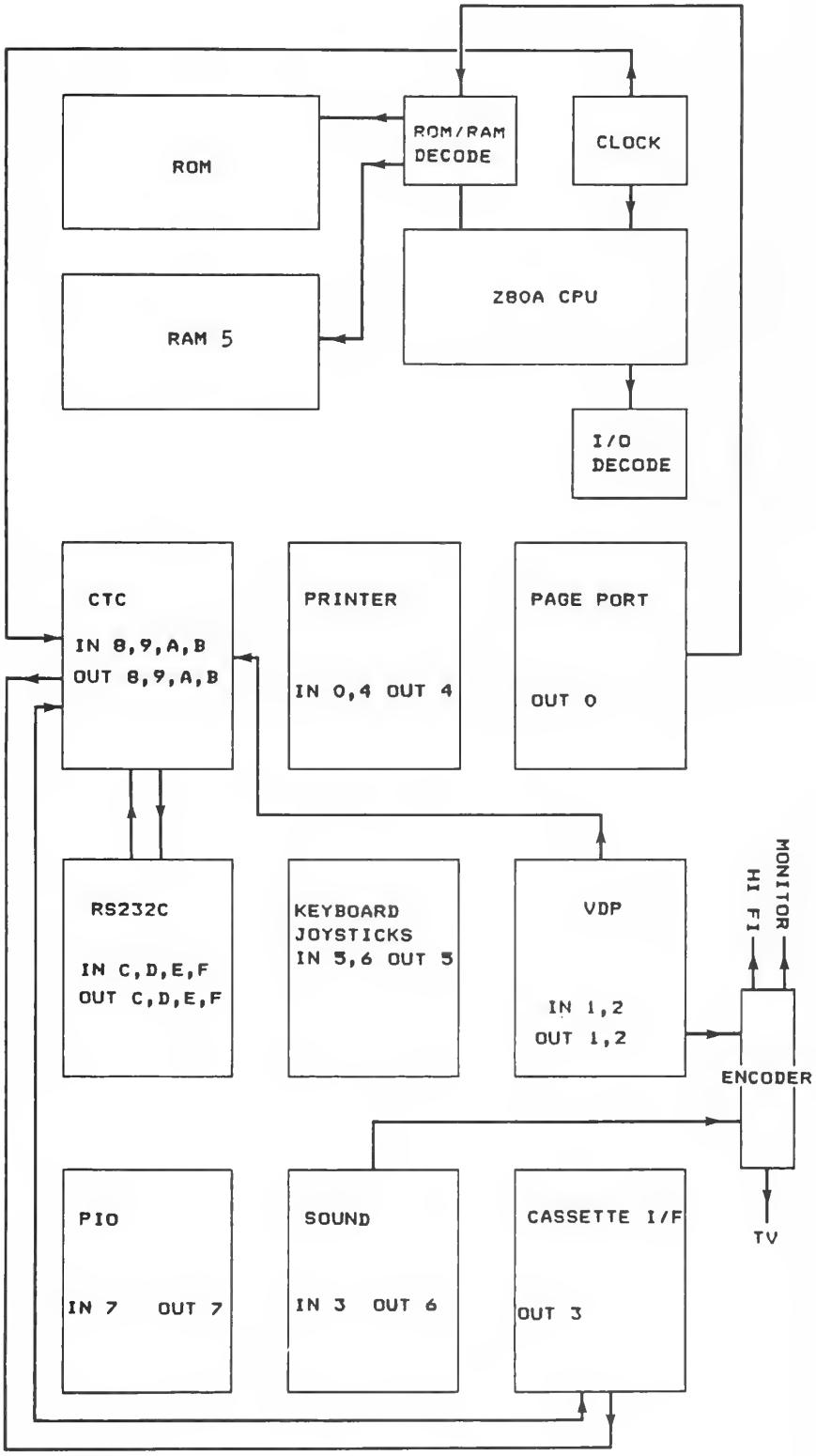
All lines are externally available on J10, which is a 60 way (30 + 30) 0.1" card edge plug, or internally on J0 which is also a 0.1" 60 way card edge plug.



NOTE :- J10 ALSO HAS KEYWAY BETWEEN 26 AND 27

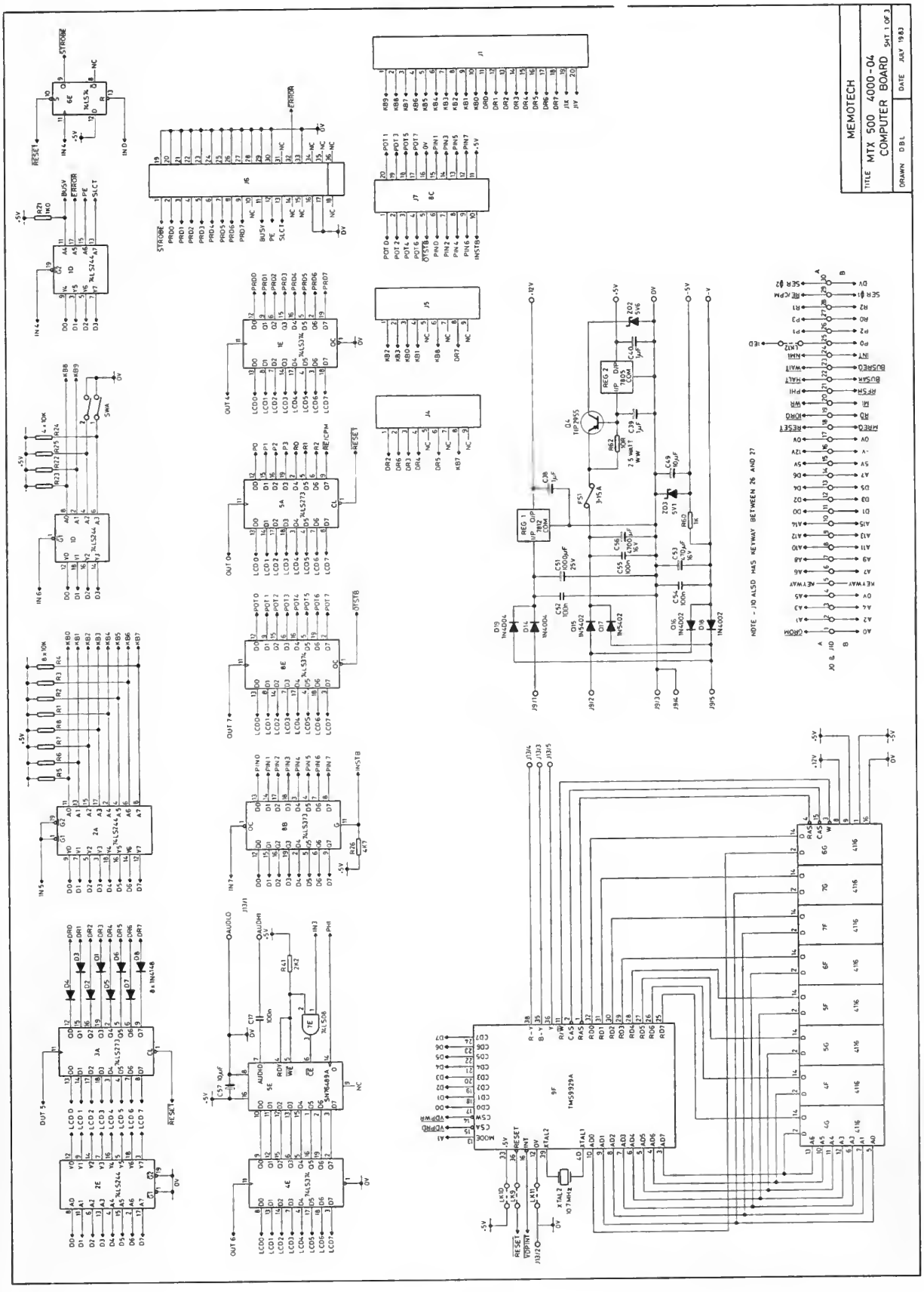
Note: (1) J10 is a mirror image of J0
(2) component side = A
Solder side = B

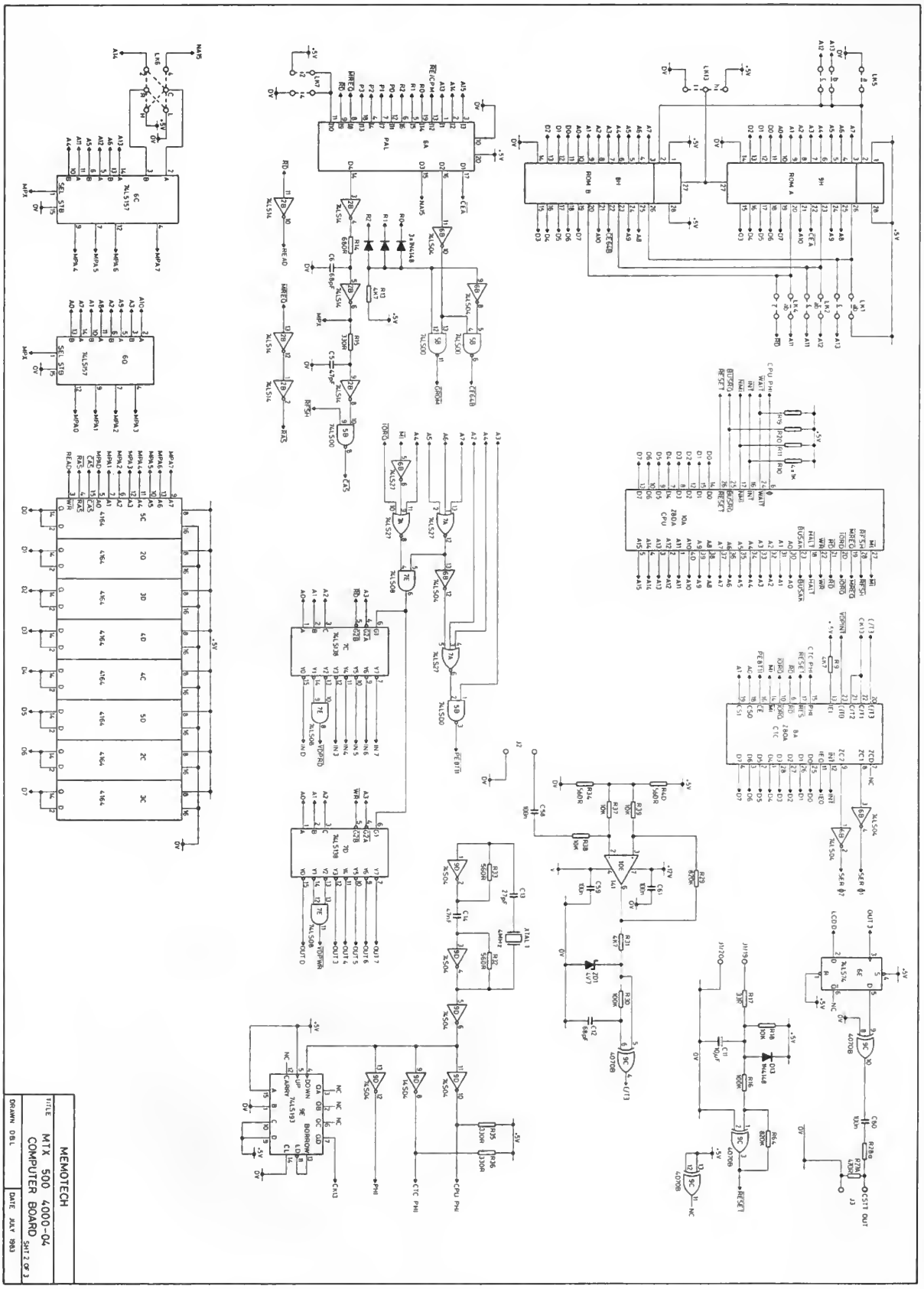
APPENDIX 4 SYSTEM BLOCK DIAGRAM



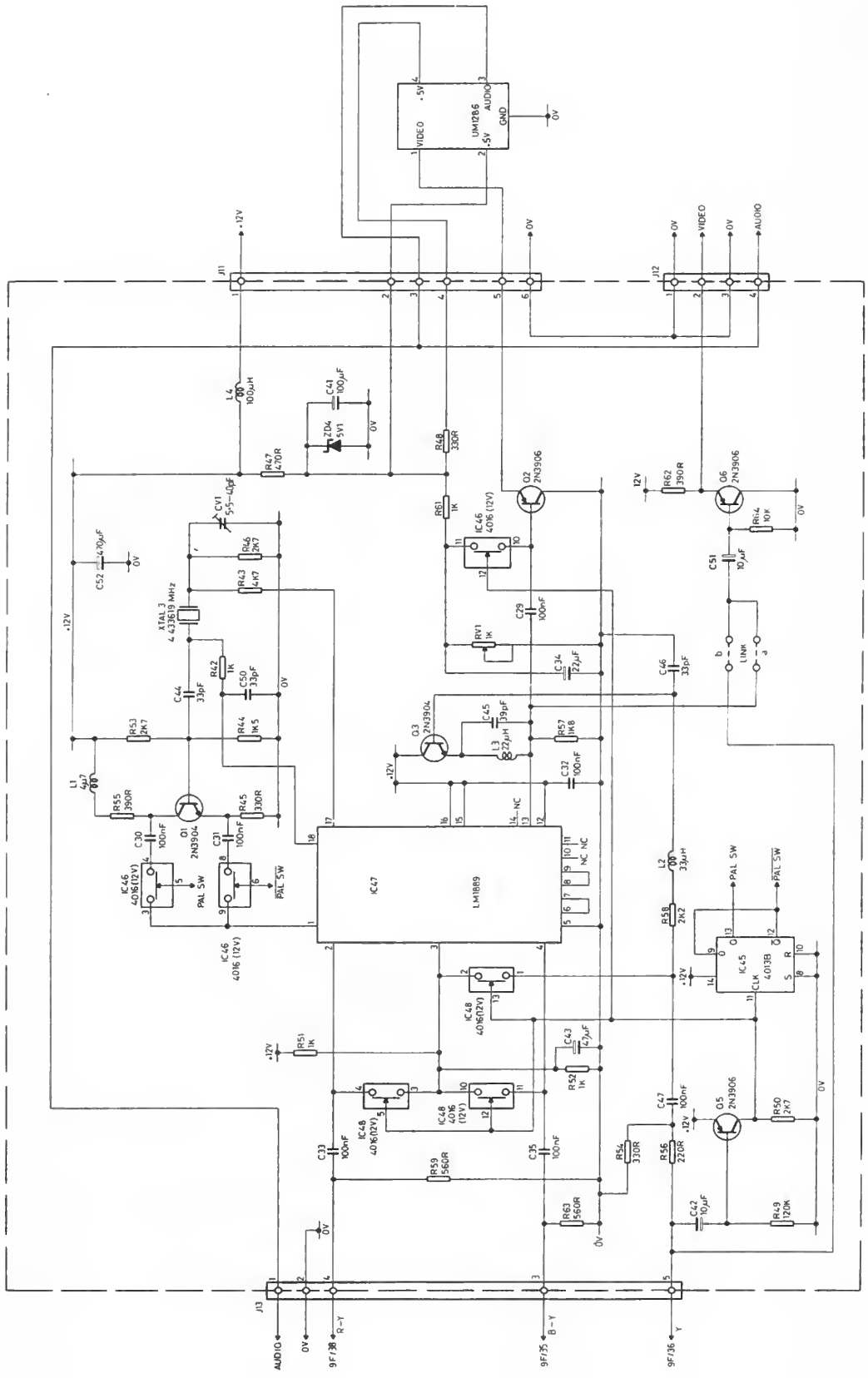
APPENDIX 5 ELECTRONIC CIRCUIT SCHEMATIC

MTX 500 4000–04

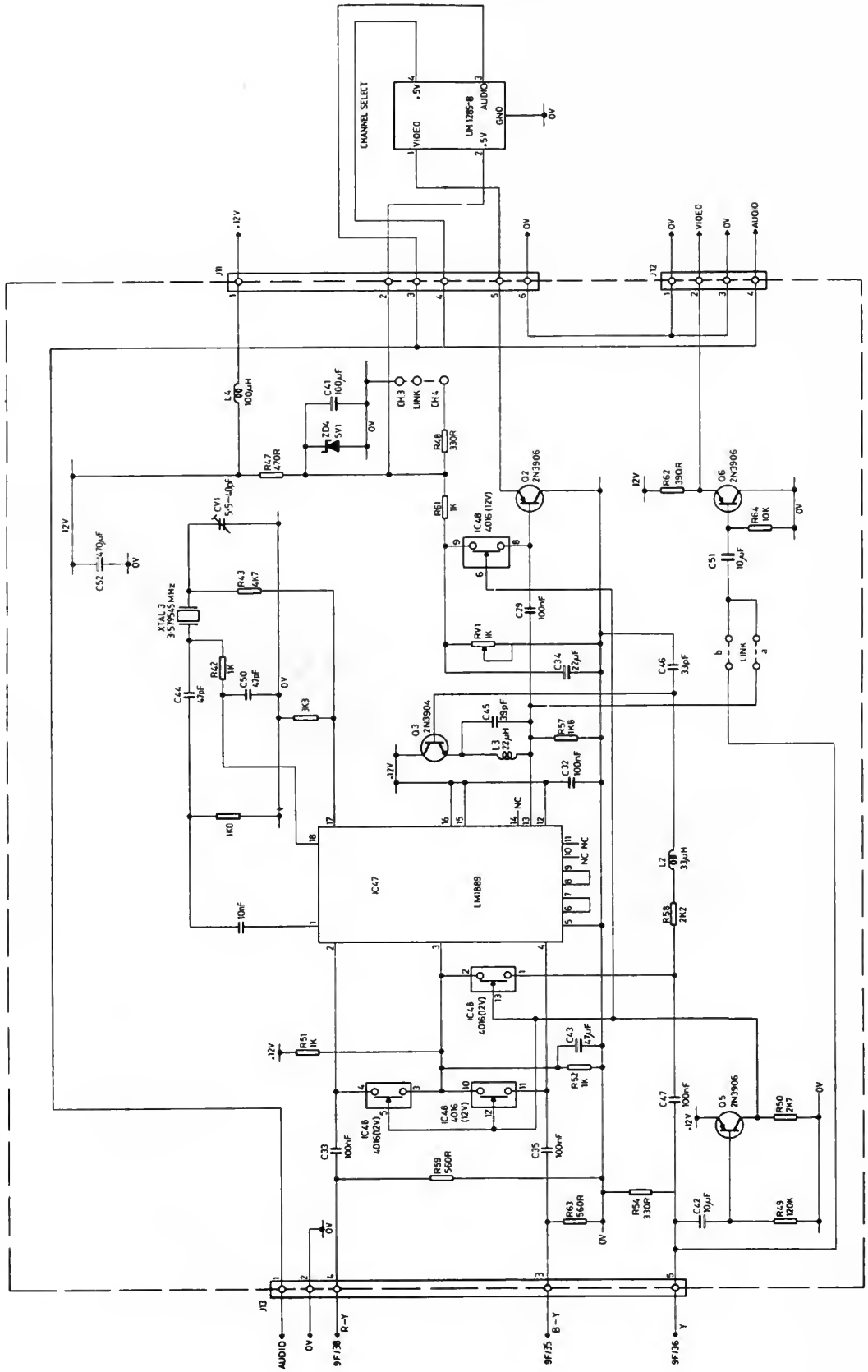



MEMOTECH
TITLE MTX 500 4000-04
COMPUTER BOARD SHT 2 OF 3
DRAWN D.B.L
DATE JULY 1983

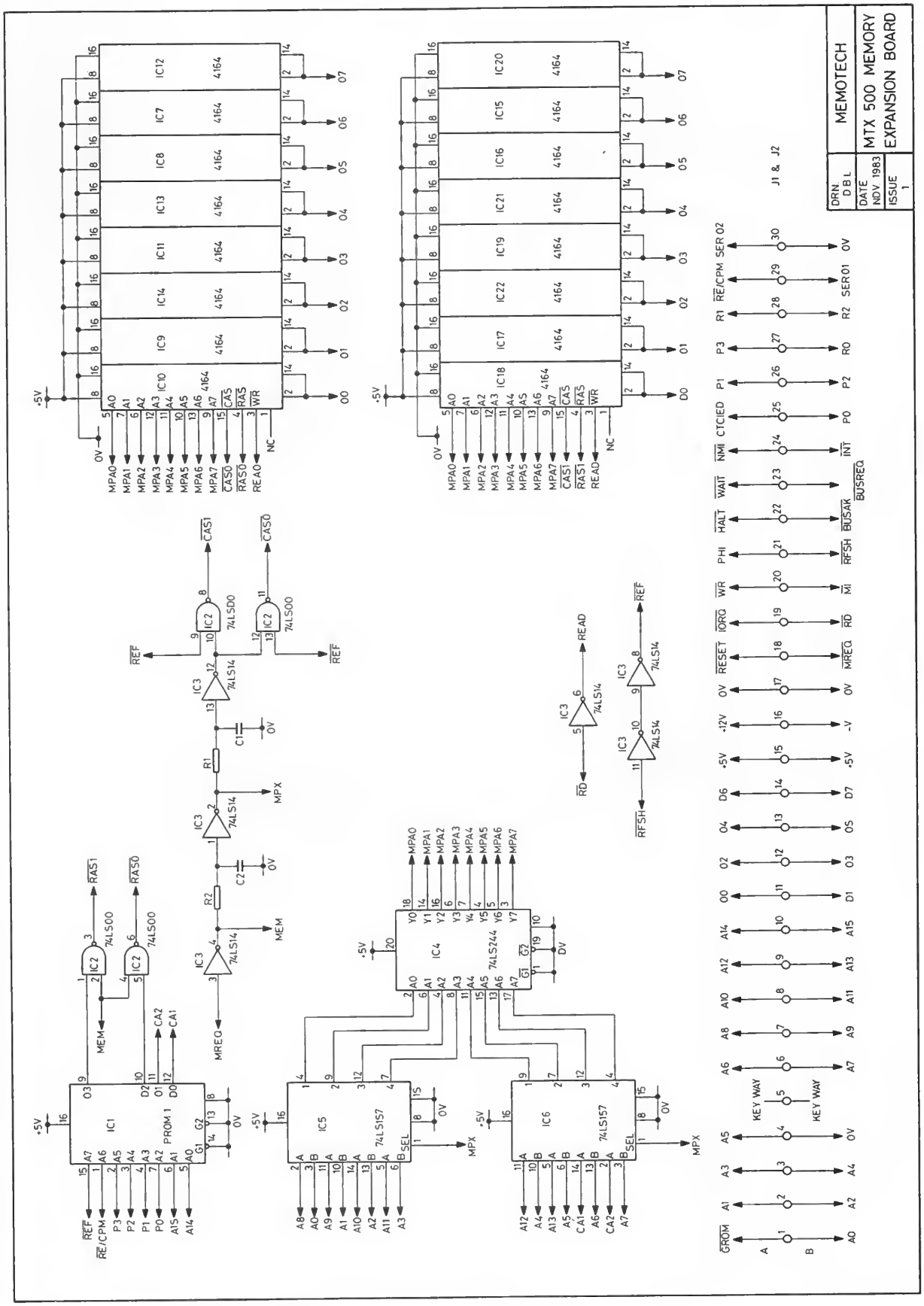
PAL BOARD

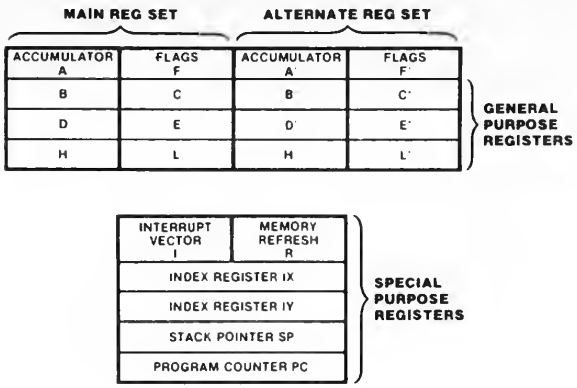

UM1286
VIDEO
+5V
-5V
AUDIO
GND
0V
+12V
J11
J12
J13
L4
100µH
C41
100µF
ZD4
5V1
R47
470R
R48
330R
Q2
2N3906
Q6
2N3906
R61
1K
IC46
4016 (12V)
R62
390R
R64
10K
C51
10µF
CV1
5-50pF
C52
470µF
R46
2K7
R43
4K7
XTAL 3
4.433619 MHz
C29
100nF
RV1
1K
C34
22µF
C46
33pF
LINK
R42
1K
C44
33pF
C50
33pF
R53
2K7
R44
1K5
Q3
2N3904
C45
39pF
L3
22µH
R57
1K8
L1
4µ7
R55
390R
R45
330R
Q1
2N3904
C32
100nF
C30
100nF
C31
100nF
PAL SW
IC47
LM1889
NC
L2
33µH
R58
2K2
IC45
4013B
CLK
R51
1K
IC48
4016(12V)
C43
4.7µF
R52
1K
C47
100nF
Q5
2N3906
R50
2K7
C33
100nF
C35
100nF
R54
330R
R56
220R
R59
560R
R63
560R
C42
10µF
R49
120K
AUDIO
0V
9F/38
R-Y
9F/35
B-Y
9F/36
Y

NTSC VIDEO BOARD



MEMORY EXPANSION BOARD





Z80-CPU INSTRUCTION SET

ADC HL, ss	Add with Carry Reg. pair ss to HL
ADC A, s	Add with carry operand s to Acc.
ADD A, n	Add value n to Acc.
ADD A, r	Add Reg. r to Acc.
ADD A, (HL)	Add location (HL) to Acc.
ADD A, (IX+d)	Add location (IX+d) to Acc.
ADD A, (IY+d)	Add location (IY+d) to Acc.
ADD HL, ss	Add Reg. pair ss to HL
ADD IX, pp	Add Reg. pair pp to IX
ADD IY, rr	Add Reg. pair rr to IY
AND s	Logical 'AND' of operand s and Acc.
BIT b, (HL)	Test BIT b of location (HL)
BIT b, (IX+d)	Test BIT b of location (IX+d)
BIT b, (IY+d)	Test BIT b of location (IY+d)
BIT b, r	Test BIT b of Reg. r
CALL cc, nn	Call subroutine at location nn if condition cc if true
CALL nn	Unconditional call subroutine at location nn
CCF	Complement carry flag
CP s	Compare operand s with Acc.
CPD	Compare location (HL) and Acc. decrement HL and BC
CPDR	Compare location (HL) and Acc. decrement HL and BC, repeat until BC=0
CPI	Compare location (HL) and Acc. increment HL and decrement BC
CPIR	Compare location (HL) and Acc. increment HL, decrement BC repeat until BC=0
CPL	Complement Acc. (1's comp)
DAA	Decimal adjust Acc.
DEC m	Decrement operand m
DEC IX	Decrement IX
DEC IY	Decrement IY
DEC ss	Decrement Reg. pair ss
DI	Disable interrupts
DJNZ e	Decrement B and Jump relative if B≠0
EI	Enable interrupts
EX (SP), HL	Exchange the location (SP) and HL
EX (SP), IX	Exchange the location (SP) and IX
EX (SP), IY	Exchange the location (SP) and IY
EX AF, AF'	Exchange the contents of AF and AF'
EX DE, HL	Exchange the contents of DE and HL
EXX	Exchange the contents of BC, DE, HL with contents of BC', DE', HL' respectively
HALT	HALT (wait for interrupt or reset)
IM 0	Set interrupt mode 0
IM 1	Set interrupt mode 1
IM 2	Set interrupt mode 2
IN A, (n)	Load the Acc. with input from device n
IN r, (C)	Load the Reg. r with input from device (C)
INC (HL)	Increment location (HL)
INC IX	Increment IX
INC (IX+d)	Increment location (IX+d)
INC IY	Increment IY
INC (IY+d)	Increment location (IY+d)
INC r	Increment Reg. r
INC ss	Increment Reg. pair ss
IND	Load location (HL) with input from port (C), decrement HL and B
INDR	Load location (HL) with input from port (C), decrement HL and decrement B, repeat until B=0
INI	Load location (HL) with input from port (C); and increment HL and decrement B
INIR	Load location (HL) with input from port (C), increment HL and decrement B, repeat until B=0
JP (HL)	Unconditional Jump to (HL)
JP (IX)	Unconditional Jump to (IX)
JP (IY)	Unconditonal Jump to (IY)
JP cc, nn	Jump to location nn if condition cc is true
JP nn	Unconditional jump to location nn
JP C, e	Jump relative to PC+e if carry=1
JR e	Unconditional Jump relative to PC+e
JP NC, e	Jump relative to PC+e if carry=0
JR NZ, e	Jump relative to PC+e if non zero (Z=0)
JR Z, e	Jump relative to PC+e if zero (Z=1)
LD A, (BC)	Load Acc. with location (BC)
LD A, (DE)	Load Acc. with location (DE)
LD A, I	Load Acc. with I
LD A, (nn)	Load Acc. with location nn
LD A, R	Load Acc. with Reg. R
LD (BC), A	Load location (BC) with Acc.
LD (DE), A	Load location (DE) with Acc.
LD (HL), n	Load location (HL) with value n
LD dd, nn	Load Reg. pair dd with value nn
LD HL, (nn)	Load HL with location (nn)
LD (HL), r	Load location (HL) with Reg. r
LD I, A	Load I with Acc.
LF IX, nn	Load IX with value nn
LD IX, (nn)	Load IX with location (nn)
LD (IX+d), n	Load location (IX+d) with value n
LD (IX+d), r	Load location (IX+d) with Reg. r
LD IY, nn	Load IY with value nn
LD IY, (nn)	Load IY with location (nn)
LD (IY+d), n	Load location (IY+d) with value n
LD (IY+d), r	Load location (IY+d) with Reg. r
LD (nn), A	Load location (nn) with Acc.
LD (nn), dd	Load location (nn) with Reg. pair dd
LD (nn), HL	Load location (nn) with HL
LD (nn), IX	Load location (nn) with IX

LD (nn), IY	Load location (nn) with IY
LD R, A	Load R with Acc.
LD r, (HL)	Load Reg. r with location (HL)
LQ r, (IX+d)	Load Reg. r with location (IX+d)
LD r, (IY+d)	Load Reg. r with location (IY+d)
LD r, n	Load Reg. r with value n
LD r, r'	Load Reg. r with Reg. r'
LD SP, HL	Load SP with HL
LD SP, IX	Load SP with IX
LD SP, IY	Load SP with IY
LDD	Load location (DE) with location (HL), decrement DE, HL and BC
LDDR	Load location (DE) with location (HL), decrement DE, HL and BC; repeat until BC=0
LDI	Load location (DE) with location (HL), increment DE, HL, decrement BC
LDIR	Load location (DE) with location (HL), increment DE, HL, decrement BC and repeat until BC=0
NEG	Negate Acc. (2's complement)
NOP	No operation
OR s	Logical 'OR' or operand s and Acc.
OTDR	Load output port (C) with location (HL) decrement HL and B, repeat until B=0
OTIR	Load output port (C) with location (HL), increment HL, decrement B, repeat until B=0
OUT (C), r	Load output port (C) with Reg. r
OUT (n), A	Load output port (n) with Acc.
OUTD	Load output port (C) with location (HL), decrement HL and B
OUTI	Load output port (C) with location (HL), increment HL and decrement B
POP IX	Load IX with top of stack
POP IY	Load IY with top of stack
POP qq	Load Reg. pair qq with top of stack
PUSH IX	Load IX onto stack
PUSH IY	Load IY onto stack
PUSH qq	Load Reg. pair qq onto stack
RES b, m	Reset Bit b of operand m
RET	Return from subroutine
RET cc	Return from subroutine if condition cc is true
RETI	Return from interrupt
RETN	Return from non maskable interrupt
RL m	Rotate left through carry operand m
RLA	Rotate left Acc. through carry
RLC (HL)	Rotate location (HL) left circular
RLC (IX+d)	Rotate location (IX+d) left circular
RLC (IY+d)	Rotate location (IY+d) left circular
RLC r	Rotate Reg. r left circular
RLCA	Rotate left circular Acc.
RLD	Rotate digit left and right between Acc. and location (HL)
RR m	Rotate right through carry operand m
RRA	Rotate right Acc. through carry
RRC m	Rotate operand m right circular
RRCA	Rotate right circular Acc.
RRD	Rotate digit right and left between Acc. and location (HL)
RST p	Restart to location p
SBC A, s	Subtract operand s from Acc. with carry
SBC HL, ss	Subtract Reg. pair ss from HL with carry
SCF	Set carry flag (C=1)
SET b, (HL)	Set Bit b of location (HL)
SET b, (IX+d)	Set Bit b of location (IX+d)
SET b, (IY+d)	Set Bit b of location (IY+d)
SET b, r	Set Bit b of Reg. r
SLA m	Shift operand m left arithmetic
SRA m	Shift operand m right arithmetic
SRL m	Shift operand m right logical
SUB s	Subtract operand s from Acc.
XOR s	Exclusive 'OR' operand s and Acc.

APPENDIX 6ii Z80 CTC PROGRAMMING SUMMARY

CTC CHANNEL INTERRUPTS WHEN 01_H IS DECREMENTED TO 00_H

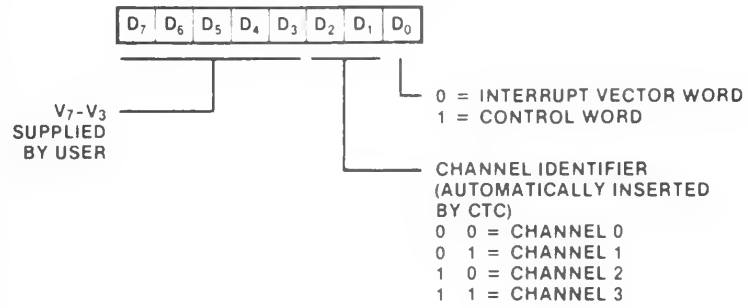
TIME CONTENT	DECIMAL COUNTS TO INTERRUPT
01_H	1
•	•
•	•
FF_H	255
00_H	256

REGISTER SELECTION

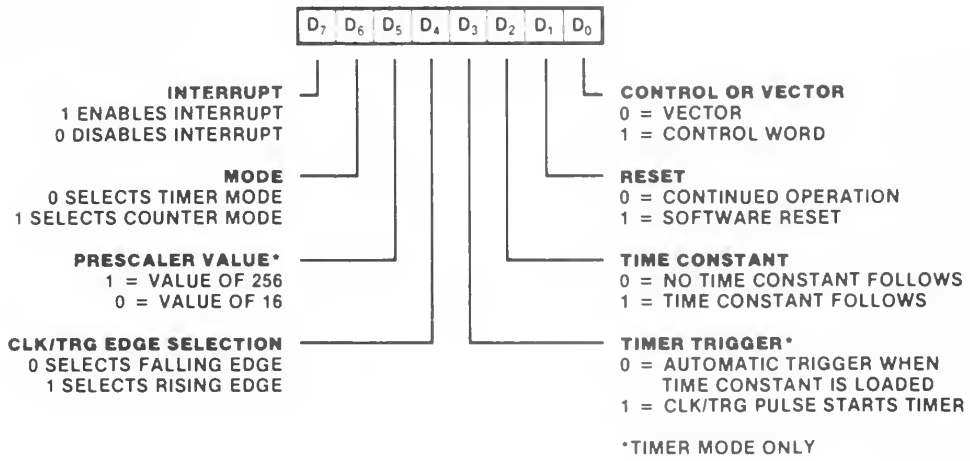
SELECT LINES CS_1	SELECT LINES CS_0	CHANNEL SELECTED	PRIORITY
0	0	0	HIGHEST
0	1	1	
1	0	2	
1	1	3	LOWEST

READ = DOWN COUNTER, WRITE = CONTROL REGISTER

Interrupt Vector Word

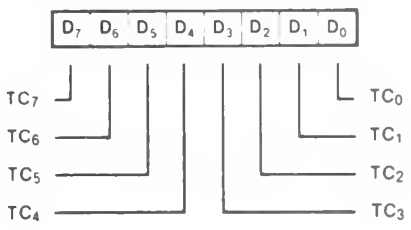


Channel Control Word



*TIMER MODE ONLY

Time Constant Word



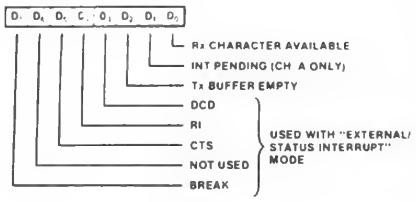
APPENDIX 6iii Z80 DART PROGRAMMING SUMMARY

CHANNEL SELECTION

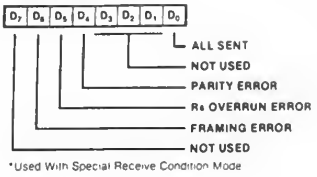
C/D	B/A	FUNCTION
0	0	CHANNEL A DATA
0	1	CHANNEL B DATA
1	0	CHANNEL A COMMANDS / STATUS
1	1	CHANNEL B COMMANDS / STATUS

Z-80 DART Read and Write Registers

READ REGISTER 0

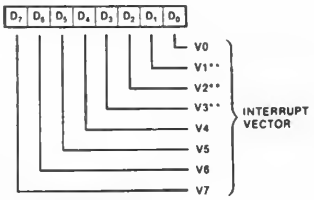


READ REGISTER 1*



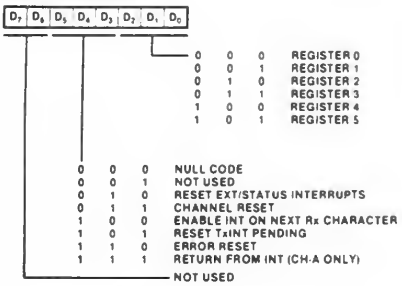
*Used With Special Receive Condition Mode

READ REGISTER 2

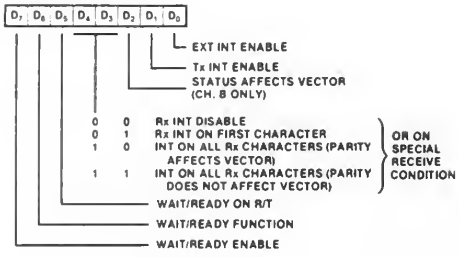


**Variable If Status Affects Vector Is Programmed

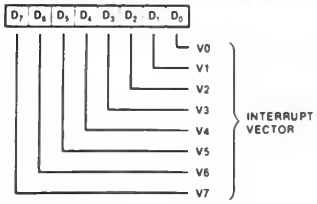
WRITE REGISTER 0



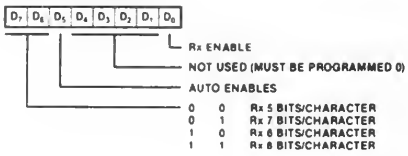
WRITE REGISTER 1



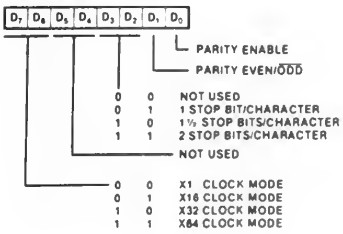
WRITE REGISTER 2 (CHANNEL B ONLY)



WRITE REGISTER 3



WRITE REGISTER 4



WRITE REGISTER 5

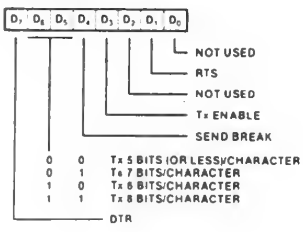
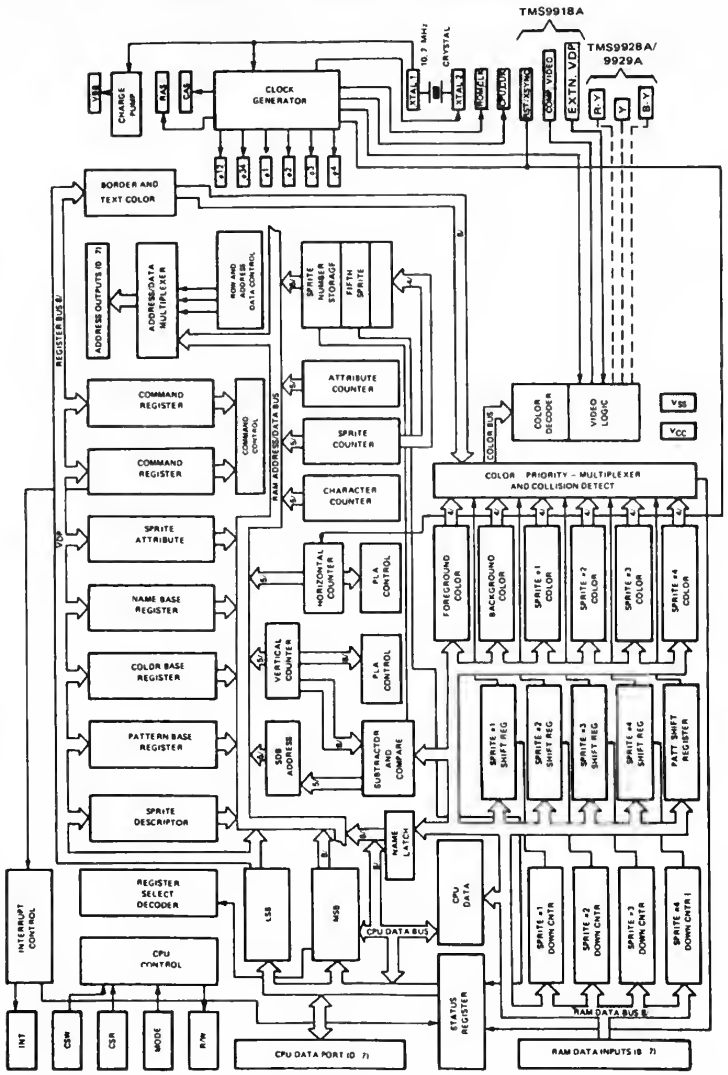


FIGURE 1 - TMS 9918 VDP BLOCK DIAGRAM



The Video Display Processor (VDP) used in the MTX Series is the TMS9918 Series. The TMS9929A is used in computers for the European market, and the TMS9928A is used for North America. The VDP is I/O mapped at ports 1 and 2. (MODE = 0 for port 1, and MODE = 1 for port 2.) The colour difference signals are encoded, mixed with sound and fed to the appropriate RF modulator, dependent upon the country for which the machine is intended.

CPU Interface Control Signals

The type and direction of data transfers are controlled by the CSW, CSR and MODE inputs. CSW is the CPU-to-VDP write select. When it is active (low), the 8 bits on D7-D0 are strobed into the VDP. CSR is the CPU-from-VDP read select. When it is active (low), the VDP outputs 8 bits on D7-D0 to the CPU. CSW and CSR should never be simultaneously low. If both are low, the VDP outputs data on D7-D0 and latches in invalid data.

MODE determines the source or destination of a read or write data transfer. MODE is normally tied to a CPU low order address line.

CPU WRITE TO VDP REGISTER

The VDP has eight write-only registers and one read-only status register. The write-only registers control the VDP operation and determine the way in which VRAM is allocated. The status register contains interrupt, sprite coincidence and fifth sprite status flags.

Each of the eight VDP write-only registers can be loaded using two 8-bit data transfers from the CAPU Table 1 describes the required format for the two bytes. The first byte transferred is the data byte, and the second byte transferred controls the destination. The most-significant bit of the second byte must be a '1'. The next four bits are '0's, and the lowest three bits make up the destination register number. The MODE input is high for both byte transfers.
To rewrite the data for an internal register after a byte of data has been loaded, the status register must be read so that internal logic will accept the next byte as data and not as a register destination. This situation may be encountered in interrupt-driven program environments. Whenever the status of VDP write parameters is in question, this procedure should be used. Note that the CPU address is destroyed by writing to the VDP register.

CPU WRITE TO VRAM

The CPU transfers data to the VRAM through the VDP using a 14-bit autoincrementing address register. Two-byte transfers are required to set up the address register. A one-byte transfer is then required to write the data to the addressed VRAM byte. The address register is then autoincremented. Sequential VRAM writes require only one byte transfer since the address register is already set up. During setup of the address register, the two most significant bits of the second address byte must be '0' and '1' respectively MODE is high for both address transfers and low for the data transfer. CSW is used in all transfers to strobe the 8 bits into the VDP. See Table 1.

TABLE 1–CPU/VDP DATA TRANSFERS

WRITE TO VDP REGISTER

OPERATION	MSB BIT 7	6	5	4	3	2	1	LSB 0	CSW	CSR	MODE
Byte 1 Data Write	D7	D6	D5	D4	D3	D2	D1	D0	0	1	1
Byte 2 Register Select	1	0	0	0	0	RS2	RS1	RS0	0	1	1

WRITE TO VRAM

Byte 1 Address set up	A7	A6	A5	A4	A3	A2	A1	A0	0	1	1
Byte 2 Address set up	0	1	A13	A12	A11	A10	A9	A8	0	1	1
Byte 3 Data Write	D7	D6	D5	D4	D3	D2	D1	D0	0	1	0

READ FROM VDP REGISTER

Byte 1	Data Read	D7	D6	D5	D4	D3	D2	D1	D0	1	0	1

READ FROM VRAM

Byte 1	Address set up	A7	A6	A5	A4	A3	A2	A1	A0	0	1	1
Byte 2	Address set up	0	0	A13	A12	A11	A10	A9	A8	0	1	1
Byte 3	Data Read	D7	D6	D5	D4	D3	D2	D1	D0	1	0	0

CPU READ FROM VDP STATUS REGISTER

The CPU can read the contents of the status register with a single-byte transfer. MODE is high for the transfer. CSR is used to signal the VDP that a read operation is required.

CPU READ FROM VRAM

The CPU reads data from VRAM through the VDP using the autoincrementing address register. A one-byte transfer is then required to read the data from the addressed VRAM byte. The address register is then autoincremented. Sequential VRAM data reads require only a one-byte transfer since the address register is already set up. During setup of the address register, the two most significant bits of the second address byte must be 0's. By setting up the address this way, a read cycle to VRAM is initiated and read data will be available for the first data transfer to the CPU. (See Table 1.) MODE is high for the address byte transfers and low for the data transfers. The VDP requires approximately 8 microseconds to fetch the VRAM byte following a data transfer and 3 microseconds following address setup.

VDP INTERRUPT

The VDP INT output pin is used to generate an interrupt at the end of each active display scan, which is about every 1/50 second (1/60 North America). The INT output is active when the Interrupt Enable bit (IE) in VDP register 1 is a '1' and the F bit of the status register is a '1'. Interrupts are cleared when the status register is read.

VDP INITIALISATION

The VDP is externally initialised whenever the RESET input is active (low) and must be held low for a minimum of 3 microseconds. The external reset synchronises all clocks with its falling edge, sets the horizontal and vertical counters to known states, and clears VDP registers 0 and 1. The video display is automatically blanked since the BLANK bit in VDP register 1 becomes a '0'. The VDP, however, continues to refresh the VRAM even though the display is blanked. While the RESET line is active, the VDP does not refresh VRAM.

VDP/VRAM INTERFACE

The VDP can access up to 16,384 bytes of VRAM using a 14-bit VRAM address. The VDP fetches data from the VRAM in order to process the video image as described later. The VDP also stores data in or reads in data from the VRAM during a CPU-VRAM data transfer. The VDP automatically refreshes the VRAM.

VRAM INTERFACE CONTROL SIGNALS

The VDP-VRAM interface consists of two unidirectional 8-bit data buses and three control lines. The VRAM outputs data to the VDP on the VRAM read data bus (RD0-RD7). The VDP outputs both the address and data to the VRAM over the VRAM address/data bus (AD0-AD7). The VRAM row address is output when RAS is active (low). The column address is output when CAS is active (low). Data is output to the VRAM when R/W is active (low).

WRITE-ONLY REGISTERS

The eight VDP write-only registers are shown in Table 2. Registers 0 and 1 contain flags to enable or disable various VDP features and modes. Registers 2 through 6 contain values that specify starting locations of various sub-blocks of VRAM. Register 7 is used to define backdrop and text colours.

REGISTER	MSB< Bit >LSB 7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	M3	EV
1	4/16K	BLANK	IE	M1	M2	0	SIZE	MAG
2	0	0	0	0	NAME TABLE BASE ADDRESS			
3	<——— COLOUR TABLE BASE ADDRESS ———>							
4	0	0	0	0	0	PATTERN GENERATOR BASE ADDRESS		
5	0	<—— SPRITE ATTRIBUTE TABLE BASE ADDRESS ——>						
6	0	0	0	0	0	SPRITE PATTERN GENERATOR BASE ADDRESS		
7	<—— TEXT COLOUR 1 ——>				TEXT COLOUR 0/BACKDROP COLOUR			
STATUS READ ONLY	F	5S	C	<——— FIFTH SPRITE NUMBER ———>				

TABLE 2. VDP REGISTERS

The following is a description of each register:

REGISTER 0 contains two VDP option control bits. All other bits are reserved for future use and must be '0's'.

BIT 1 M3 (mode bit 3).
BIT 0 External Video enable/disable
'1' enables external video input
'0' disables external video input

REGISTER 1 contains 8 VDP option control bits.

BIT 7 7 4/16k selection
'0' selects 4K RAM operation
'1' selects 16K RAM operation (MTX operation)

BIT 6 BLANK enable/disable
'0' causes the active display area to blank
'1' enables the active display
Blanking causes the display to show border colour only.

BIT 5 IE (Interrupt Enable)
'0' disable VDP interrupt
'1' enable VDP interrupt

BIT 4,3 M1,M2 (mode bits 1 and 2)
M1,M2 and M3 determine the operating mode of the VDP:

M1	M2	M3	
0	0	0	Graphics I mode
0	0	1	Graphics II mode
0	1	0	Multicolour mode
1	0	0	Text mode

BIT 2 Reserved

BIT 1 Size (sprite size select)
'0' selects Size 0 sprites (8 x 8 bits)
'1' selects Size 1 sprites (16 x 16 bits)

BIT 0 MAG (Magnification option for sprites)
'0' selects MAG0 sprites (1x)
'1' selects MAG1 sprites (2x)

REGISTER 2 defines the base address of the Name Table sub-block. The range on its contents is from 0 to 15. The contents of the register form the upper 4 bits of the 14-bit Name Table addresses; thus the Name Table base address is equal to (register 2) * 400h.

REGISTER 3 defines the base address of the Colour Table sub-block. The range on its contents is from 0 to 255. The contents of the register form the upper 8 bits of the 14-bit Colour Table addresses; thus the Colour Table base address is equal to (register 3) *40h.

REGISTER 4 defines the base address of the Pattern, Text or Multicolour Generator sub-block. The range of its contents is 0 through 7. The contents of the register form the upper 3 bits of the 14-bit Generator addresses; thus the Generator base address is equal to (register 4)* 800h.

REGISTER 5 defines the base address of the Sprite Attribute Table sub-block. The range of its contents is from 0 through 127. The contents of the register form the upper 7 bits of the 14-bit Sprite Attribute Table addresses; thus the base address is equal to (register 5) *80h.

The VDP registers define the base addresses for several sub-blocks within VRAM. These sub-blocks form tables which are used to produce the desired image on the TV screen. The Pattern Name Table, the Pattern Generator Table and the Sprite Generator Table are used to form the sprites. The contents of these tables must all be provided by the microprocessor. Animation is achieved by altering the contents of VRAM in real time.

The VDP can display the 15 colours, plus transparent shown in Table 3. The VDP colours also provide eight different grey levels for displays on monochrome televisions; the luminance values in the table indicate these levels, 0.00 being black and 1.00 being white. Whenever all planes are of the transparent colour at a given point, the colour shown at that point will be black.

REGISTER 6 defines the base address of the Sprite Pattern Generator sub-block. The range of its contents is 0 through 7. The contents of the register form the upper 3 bits of the 14-bit Sprite Pattern Generator addresses, thus the sprite Pattern Generator base address is equal to (register 6) *800h

REGISTER 7 The upper 4 bits contain the colour code of colour 1 in the Text mode. The lower 4 bits contain the colour code for colour 0 in the Text mode and the backdrop colour in all modes. See Table 4 for colour codes.

STATUS REGISTER

The VDP has a single 8-bit status register that can be accessed by the CPU. The status register contains the interrupt pending flag, the sprite coincidence flag, the fifth sprite flag, and the fifth sprite number, if one exists. The format of the status register is shown in Table 2. A discussion of the contents follows.
The status register may be read at any time to test the F, C, and 5S status bits. Reading the status register will clear the interrupt flag, F. Asynchronous reads will, however, cause the frame flag (F) bit to be reset and therefore missed. Consequently, the status register should be read only when the VDP interrupt is pending.

INTERRUPT FLAG (F)

The F status flag in the status register is set to '1' at the end of the raster scan of the last line of the active display. It is reset to a '0' after the status register is read or when the VDP is externally reset. If the Interrupt Enable bit in VDP register 1 is active ('1'), the VDP interrupt output (INT) will be active (low) whenever the F status flag is a '1'.

COINCIDENCE FLAG (C)

The C status flag in the status register is set to a '1' if two or more sprites "coincide". Coincidence occurs if any two sprites on the screen have one or more overlapping pixels. Transparent colour sprites, as well as those that are partially or completely off the screen, are also considered. Sprites beyond the Sprite Attribute Table terminator (D016) are not considered. The 'C' flag is cleared to a '0' after the status register is read or the VDP is externally reset.

FIFTH SPRITE FLAG (5S) AND NUMBER

The 5S status Flag in the status register is set to a '1' whenever there are five or more sprites on a horizontal line (lines 0 to 192) and the frame flag is equal to a '0'. The 5S status flag is cleared to a '0' after the status register is read or the VDP is externally reset. The number of the fifth sprite is placed into the fower 5 bits of the status register when the 5S flag is set and is valid whenever the 5S flag is '1'. The setting of the fifth sprite flag will not cause an interrupt. The VDP operates at 262 lines per frame and approximately 60 frames per second in a non-interlaced mode of operation.

TABLE 3–SCREEN DISPLAY PARAMETERS

PARAMETER	PIXEL CLOCK CYCLES	
HORIZONTAL	PATTERN/MULTICOLOUR	TEXT
Horizontal Active Display	256	240
Right Border	15	25
Right Blanking	8	8
Horizontal Sync	26	26
Left Blanking	2	2
Colour Burst	14	14
Left Blanking	8	8
Left Border	13	19
	342	342

VERTICAL	LINE
Vertical Active Display	192
Bottom Border	24
Bottom Blanking	3
Vertical Sync	3
Top Blanking	13
Top Border	27
	262

VIDEO DISPLAY BOARDS

The VDP displays an image on the screen that can best be envisaged as a set of display planes sandwiched together. Figure 2 shows the definition of each of the planes. Objects on planes closest to the viewer have higher priority. In cases where two entities on two different planes are occupying the same spot on the screen, the entity on the higher priority plane will show at that point. For an entity on a specific plane to show through, all planes in front of that plane must be transparent at that point. The first 32 planes each may contain a single sprite. (Sprites are pattern objects whose positions on the screen are defined by horizontal and vertical co-ordinates in VRAM.) The areas of the Sprite Planes, outside the sprite itself, are transparent. Since the co-ordinates of the sprite are in terms of pixels, the sprite can be positioned and moved about very accurately. Sprites are available in three sizes: 8 X 8 pixels, 16 X 16 pixels, and 32 X 32 pixels. Behind the Sprite Plane is the Pattern Plane. The Pattern Plane is used for textual and graphics images generated by the Text, Graphics I, Graphics II, or Multicolour modes. Behind the Pattern Plane is the backdrop, which is larger in area than the other planes so that it forms a border around them. The last and lowest priority plane is the External Video Plane. Its image is defined by the external video input pin. The backdrop consists of a single colour used for the display borders and as the default colour for the active display area. The default colour is stored in the VDP register 7. When the backdrop colour register contains the transparent code, the backdrop automatically defaults to black if the external video mode is not selected.

The 32 Sprite Planes are used for the 32 sprites in the Multicolour and Graphics modes. They are not used in the Text mode and are automatically transparent. Each of the sprites can cover an 8 X 8, 16 X 16, or 32 X 32 pixel area on its plane. Any part of the plane not covered by the sprite is transparent. All or part of each sprite may also be transparent. Sprite 0 is on the outside or highest plane, and sprite 31 is on the plane immediately adjacent to the Pattern Plane. Whenever a pixel in a Sprite Plane is transparent, the colour of the next plane can be seen through that plane. If, however, the sprite pixel is non-transparent, the colours of the lower planes are automatically replaced by the sprite colour. There is also a restriction on the number of sprites on a line. Only four sprites can be active on any horizontal line. Additional sprites on a line will be automatically made transparent for that line. Only those sprites that are active on the display will cause the coincidence flag to set. The VDP status register provides a flag bit and the number of the fifth sprite whenever this occurs. The Pattern Plane is used in the Text, Multicolour, and Graphics modes for display of the graphic patterns of characters. Whenever a pixel on the Pattern Plane is non-transparent, the backdrop colour is automatically replaced by the Pattern Plane colour. When a pixel in the Pattern Plane is transparent, the backdrop colour can be seen through the Pattern Plane. (see Fig 2 overleaf)

FIGURE 2. VDP DISPLAY PLANES

The VDP has four video colour display modes that appear on the Pattern Plane: Graphics I mode, Graphics II mode, Text mode, and Multicolour mode. Graphics I and Graphics II modes cause the Pattern Plane to be broken up into groups of 8 X 8 pixels, called pattern positions. Since the full image is 256 X 192 pixels, there are 32 X 24 pattern positions on the screen in the graphics modes. In Graphics I mode, 256 possible patterns may be defined for the 768 pattern positions with two unique colours allowed for each pattern definition. Graphics II mode provides, through a unique mapping scheme, 768 pattern definitions for the 768 pattern positions. Graphics II mode also allows the selection of two unique colours for each line of a pattern definition. Thus, all 15 colours plus transparent may be used in a single pattern position. In Text mode, the Pattern Plane is broken into groups of 6 X 8 pixels, called text positions. There are 40 X 24 text positions on the screen in this mode. In Text mode, sprites do not appear on the screen and two colours are defined for the entire screen. In Multicolour mode, the screen is broken into a grid of 64 X 48 positions, each of which is a 4 X 4 pixel. Within each position, one unique colour is allowed.

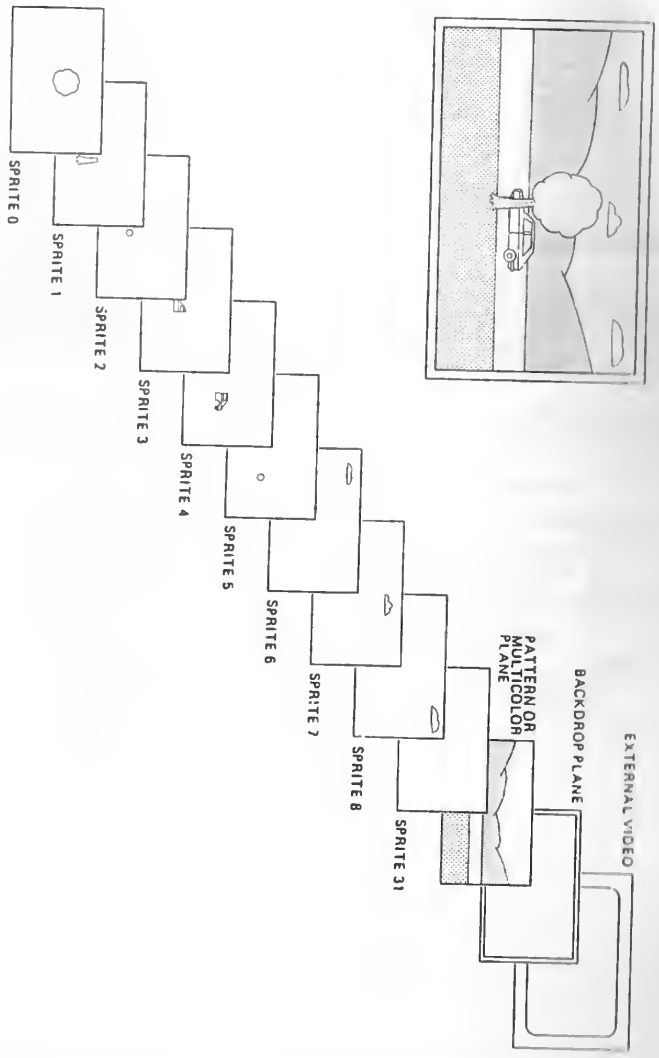


TABLE 4. Colour Assignments

COLOUR (HEX)	COLOUR	LUMINANCE (DC VALUE)	CHROMINANCE (AC VALUE)
0	Transparent	0.00	-
1	Black	0.00	-
2	Medium Green	.60	.60
3	Light Green	.80	.53
4	Dark Blue	.47	.73
5	Light Blue	.67	.60
6	Dark Red	.53	.53
7	Cyan	.80	.73
8	Medium Red	.67	.73
9	Light Red	.80	.73
A	Dark Yellow	.87	.53
B	Light Yellow	1.00	.40
C	Dark Green	.47	.60
D	Magenta	.60	.47
E	Grey	-	-
F	White	1.00	-

Graphics 1 Mode

The VDP is in Graphics 1 mode when M1, M2, and M3 bits in VDP registers 1 and 0 are zero. In Graphics 1 mode the Pattern Plane is divided into a grid of 32 columns by 24 rows of pattern positions. Each of the pattern positions contains 8 X 8 pixels. The table in VRAM is used to generate the Pattern Plane. A total of 2848 VRAM bytes are required for the Pattern Name, Colour and Generator tables. Less memory is required if all 256 possible pattern definitions are not required. The tables can be overlapped to reduce the amount of VRAM needed for pattern generation.

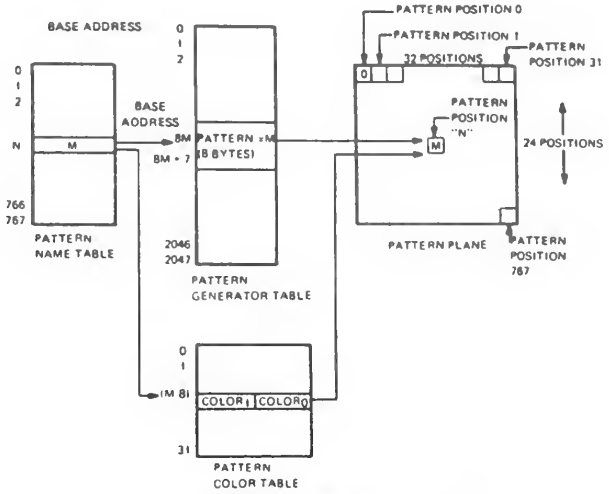


FIGURE 2-8 – PATTERN MODE MAPPING

The Pattern Generator Table contains a library of patterns that can be displayed in the pattern positions. It is 2048 bytes long, and is arranged into 256 patterns, each of which is eight bytes long, yielding 8 X 8 bits. All of the '1's in the eight-byte pattern can designate one colour (colour 1), while all the '0's can designate another colour (colour 0)

The full 8-bit pattern name is used to select one of the 256 pattern definitions in the Pattern Generator Table. The table is a 2048-byte block in VRAM beginning on a 2 kilobyte boundary. The starting address of the table is determined by the generator base address in VDP register 4. The base address forms the three most significant bits of the 14-bit VRAM address for each Pattern Generator Table entry. The next 8 bits indicate the 8-bit name of the selected pattern definition. The lowest 3 bits of the VRAM address indicate the row number within the pattern definition

Eight bytes are required for each of the 256 possible unique 8 x 8 pattern definitions. The first byte defines the first row of the pattern, and the second byte defines the second row. The first bit of each of the eight bytes define the first column of the pattern. The remaining rows and columns are similarly defined. Each bit entry in the pattern definition selects one of the two colours for that pattern. A '1' bit selects the colour code (colour 1) contained in the most significant four bits of the corresponding colour table byte. A '0' bit selects the other colour code (colour 0). An example of pattern definition mapping is provided below

Row/byte	Column 0	1	2	3	4	5	Bit 0	1	2	3	4	5	6	7
0	*	*	*	*	*		0	1	1	1	1	1	0	0
1					*		0	0	0	0	0	1	0	0
2		*	*	*	*		0	0	1	1	1	1	0	0
3					*		0	0	0	0	0	1	0	0
4					*		0	0	0	0	0	1	0	0
5					*		0	0	0	0	0	1	0	0
6	*	*	*	*	*		0	1	1	1	1	1	0	0
7							0	0	0	0	0	0	0	0
	PATTERN						PATTERN DEFINITION							

The colour of the '1's and '0's is defined by the Pattern Colour lTable that contains 32 entries each of which is one byte long. Each entry defines two colours. the most significant 4 bits of each entry define the colour of the '1's. and the least significant 4 bits define the colour of the '0's. The first entry in the colour table defines the colours for patterns 0 to 7. the next entry for patterns 8 to 15, and so on. (See Table 4 for assignments.) Thus. 32 different pairs of colours may be displayed simultaneously.

The Pattern Name Table is located in a contiguous 768-byte block in VRAM beginning on a 1 kilobyte boundary. The starting address of the Name Table is determined by the 4-bit Name Tablebase address field in VDP register 2. The base address forms the upper four bits of the 14-bit VRAM address. The lower 10 bits of the VRAM address are formed from the row and column counters.

Each byte entry in the Name Table is the name of or the pointer to a pattern definition in the Pattern Generator Table. The upper five bits of the eight-bit name identify the colour group of the pattern. There are 32 groups of eight patterns. The same two colours are used for all eight patterns in a group; the colour codes are stored in the VDP Colour Table. The Colour Table is located in a 32-byte block in VRAM beginning on a 64-byte boundary. The table starting address is determined by the 8-bit Colour Tablebase address in VDP register 3. The base address forms the upper eight bits of the 14-bit Colour Table entry VRAM address. The next bit is a '0' and the lowest 5 bits are equal to the upper 5 bits of the corresponding Name Table entries.

Since the tables in VRAM have their base addresses defined by the VDP registers, a complete switch of the values in the tables can be made by simply changing the values in the VDP registers. This is especially useful when one wishes to time slice between two or more screens of graphics.

When the Pattern Generator Table is loaded with a pattern set, manipulation of the Pattern Name Table contents can change the appearance of the screen. Alternatively, a dynamically changing set of patterns throughout the course of a graphics session is easily accomplished since all tables are in VRAM.

For textual applications, the desired character set is typically loaded into the Pattern Generator first. The official US ASCII character set might be loaded into the Pattern Generator in such a way that the pattern numbers correspond to the 8-bit ASCII codes for that pattern; e.g., the pattern for the letter "A" would be loaded into pattern number 4116 in the Pattern Generator. Next the Pattern Colour Table would be loaded up with the proper colour set. To print a textual message on the screen, write the proper ASCII codes out to the Pattern Name Table.

Images can be formed using the Pattern Plane. To display an object of size 8 x 8 pixels or smaller, only one pattern would need to be defined. To display a larger figure, the figure should be broken up into smaller 8 x 8 squares. Then multiple patterns can be defined, and the Pattern Generator and Pattern Name Table set up appropriately. Note that rough motion of objects requires merely updating entries in the Pattern Name Table.

TABLE 5 Pattern colour table

BYTE No.	PATTERN No.
0	0..7
1	8..15
2	16.23
3	24..31
4	32..39
5	40..47
6	48..55
7	56..63
8	64..71
9	72..79
10	80..87
11	88..95
12	96..103
13	104..111
14	112..119
15	120..127
16	128..135
17	136..143
18	144..151
19	152..159
20	160..167
21	168..175
22	176..183
23	184..191
24	192..199
25	200..207
26	208..215
27	216..223
28	224..231
29	232..239
30	240..247
31	248..255

A total of 2848 VRAM bytes are required for the Pattern, Name, Colour and Generator tables. Less memory is needed if all 256 possible pattern definitions are not required; the tables can be overlapped to reduce the amount of VRAM needed for pattern generation.

Graphics II Mode

The VDP is in the Graphics II mode when mode bits M1 = 0, M2 = 0, and M3 = 1. The Graphics II mode is similar to Graphics I mode except it allows a larger library of patterns so that a unique pattern generator entry may be made for each of the 768 (32 x 24) pattern positions on the video screen. Additionally, more colour information is included in each 8 x 8 graphics pattern. Thus two unique colours may be specified for each byte of the 8 x 8 pattern. A larger amount of VRAM (12 kilobytes) is required to implement the full usage of the Graphics II mode.

Like Graphics I mode, the Graphics II mode Pattern Name Table contains 768 entries which correspond to the 768 pattern positions on the display screen. Because the Graphics 1 mode pattern names are only 8 bits in length, a maximum of 256 pattern definitions may be addressed using the addressing scheme discussed in the previous section. Graphics II mode, however, segments the display screen into three equal parts of 256 pattern positions each, and also segments the Pattern Generator Table into three equal blocks of 2048 bytes each. Pattern definitions in the first third correspond to pattern positions in the upper third of the display screen. Likewise pattern definitions in the second and third blocks of the Pattern Generator Table correspond to the second and third areas of the Pattern Plane. The Pattern Name Table is also segmented into three blocks of 256 names each so that names found in the upper third, reference pattern definitions found in the upper 2048 bytes in the Pattern Generator Table. Likewise the second and third blocks reference pattern definitions in the second 2048 byte block and third 2048 byte block respectively. Thus, if 768 patterns are uniquely specified an 8-bit pattern name will be used three times, once in each segment of the Pattern Name Table. The Pattern Generator Table falls on eight kilobyte boundaries and may be located in the upper or lower half of 16K memory based on the MSB of the pattern generator base in VDP register 4. The LSB's must be set to all '1's.

The Colour Table is also 6144 bytes long and is segmented into three equal blocks of 2048 bytes. Each entry in the Pattern Colour Table is eight bytes which provides the capability to uniquely specify colour 1 and colour 0 for each of the eight bytes of the correspoonding pattern definition. The addressing scheme is exactly like that of the Pattern Generator Table except for the location of the table in VRAM. This is controlled by the loading of the MSB of the colour base in VDP register 3. The LSB's must be set to all '1's.

Figure 2-10 is an example of the Graphics II mode mapping scheme. Note that pattern names P1,P2,P3 correspond to pattern generator entries in the three blocks of the Pattern Generator Table. Note also how these three names map to the display screen.

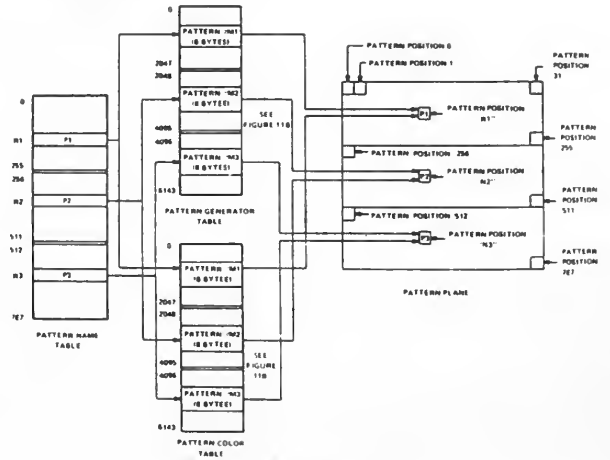


FIGURE 2-10 – GRAPHICS II MODE MAPPING

Multicolour Mode

The VDP is in Multicolour mode when mode bits M1 = 0, M2 = 1, and M3 = 0. Multicolour mode provides an unrestricted 64 x 48 colour square display. Each colour square contains a 4 x 4 block of pixels. The colour of each of the colour squares can be any one of the 15 video display colours plus transparent. Consequently, all 15 colours can be used simultaneously in the Multicolour mode. The Backdrop and Sprite Planes are still active in the Multicolour mode.

The Multicolour Name Table is the same as that for the graphics modes, consisting of 768 name entries. The name no longer points to a colour list; rather colour is now derived from the Pattern Generator Table. The name points to an eight-byte segment of VRAM in the Pattern Generator Table.

Only two bytes of the eight-byte segment are used to specify the screen image. These two bytes specify four colours, each colour occupying a 4 x 4 pixel area. The four MSB's of the first byte define the colour of the upper left quarter of the multicolour pattern; the LSB's define the colour of the upper right quarter. The second byte similarly defines the lower left and right quarters of the multicolour pattern. The two bytes thus map into a 8 x 8 pixel multicolour pattern.

The location of the two bytes within the eight-byte segment pointed to by the name is dependent upon the screen position where the name is mapped. For names in the top row (names 0-31), the two bytes are the first two within the groups of eight-byte segments pointed to by the names. The next row of names (32-63) uses the third and fourth bytes within the eight-byte segments. The next row of names uses the fifth and sixth bytes while the last row of names uses the seventh and eighth. This series repeats for the remainder of the screen.

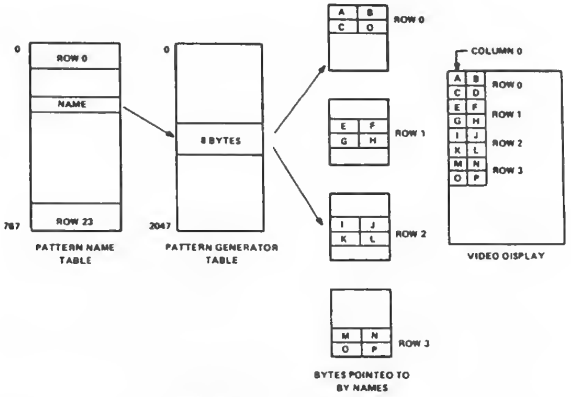


FIGURE 2.14 – MULTICOLOR MODE MAPPING

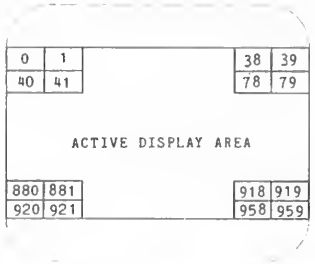
The mapping of VRAM contents to screen image is simplified by using duplicate names in the Name Table. Since the series of bytes used within the eight-byte segment repeats every four rows, the four rows in the same column can use the same name. Then the eight-byte segment specifies a 2 x 8 colour square pattern on the screen as a straightforward translation from the eight-byte segment in VRAM pointed to by the common name.

When used in this manner, 768 bytes are still used for the Name Table and 1536 bytes are used for the colour information in the Pattern Generator Table (24 rows x 32 columns x 8 bytes/pattern position). Thus a total of 1728 bytes in VRAM are required. It should be noted that the tables begin on even 1K and 2K boundaries and are therefore not contiguous.

Text Mode

The VDP is in Text mode when mode bits M1 = 1, M2 = 0, and M3 = 0. In the Text mode, the screen is divided into a grid of 40 text positions across and 24 down. Each of the text positions contains six pixels across and eight pixels down. The tables used to generate the Pattern Plane are the Pattern Name Table and the Pattern Generator Table. There can be up to 256 unique patterns defined at any time. The pattern definitions are stored in the Pattern Generator Table in VRAM and can be dynamically changed. The VRAM contains a Pattern Name Table which maps the pattern definitions into each of the 960 pattern cells on the Pattern Plane. Sprites are not available in Text mode.

TEXT MODE NAME TABLE PATTERN POSITIONS



As in the case of the Graphics modes, the Pattern Generator Table contains a library of text patterns that can be displayed in the text positions. It is 2048 bytes long, and is arranged in 256 text patterns, each of which is eight bytes long. Since each text position on the screen is only six pixels across, the least significant 2 bits of each text pattern are ignored, yielding 6 x 8 bits in each text pattern. Each block of eight bytes defines a text pattern in which all the '1's in the text pattern take on one colour when displayed on the screen, while all the '0's take on another colour. These colours are chosen by loading VDP register 7 with the colour 1 and colour 0 in the left and right nibbles respectively.

In the Text mode, the Pattern Name Table determines the position of the text pattern on the screen. There are 960 entires in the Pattern Name Table, each one byte long. There is a one-to-one correspondence between text pattern positions on the screen and entries in the Pattern Name Table (40*24 = 960). The first 40 entries correspond to the top row of text pattern positions on the screen, the next forty to the second row, and so on. The value of an entry in the Pattern Name Table indicates which of the 256 text patterns is to be placed at that spot on the Pattern Plane. The Pattern Name Table is located in a contiguous 960-byte block in VRAM beginning on a 1 kilobyte boundary. The starting address of the name table is determined by the 4-bit Name Table base address field in VDP register 2. The base address forms the upper 4 bits of the 14-bit VRAM address. The lower 10 bits of the VRAM address point to one of 960 pattern cells. The name table is organised by rows. Each byte entry in the name table is the pointer to a pattern definition in the Pattern Generator Table. The same two colours are used for all 256 patterns; the colour codes are stored in VDP register 7.

As its name implies, the Text mode is intended mainly for textual applications, especially those in which the 32 patterns per line in Graphics modes is insufficient. The advantage is that eight more patterns can be fitted onto one line; the disadvantages are that sprites cannot be used, and only two colours are available for the entire screen. With care, the same text pattern set that is used in Text mode can be also used in Graphics I mode. This is done by ensuring that the least significant 2 bits of all the character patterns are '0'. A switch from Text mode to Pattern mode, then, results in a stretching of the space between characters, and a reduction of the number of characters per line from 40 to 32. As with the Graphics Modes, once a character set has been defined and placed into the Pattern Generator, updating the Pattern Name Table will produce and manipulate textual material on the screen.

The full 8-bit pattern name is used to select one of the 256 pattern definitions in the pattern generator table. The table is a 2048-byte block in VRAM beginning on a 2 kilobyte boundary. The starting address of the table is determined by the generator base address in VDP register 4. The base address forms the 3 most significant bits of the 14-bit VRAM address for each Pattern Generator Table entry. The next 8 bits are equal to the 8-bit name of the selected pattern definition. The lowest 3 bits of the VRAM address are equal to the row number within the pattern definition.

Eight bytes are required for each of the 256 possible unique 6 x 8 pattern definitions. The first byte defines the first row of the pattern, and the second byte defines the second row. The two least significant bits in each byte are not used. It is, however, strongly recommended that these bits be '0's. Each bit entry in the pattern definition selects one of the two colours for that pattern. A '1' bit selects the colour code (colour 1) contained in the most significant 4 bits of VDP register 7. A '0' bit selects the other colour code (colour 0) which is in the least significant 4 bits of the same VDP Register.

A total of 3005 VRAM bytes are required for the Pattern Name and Generator Tables. Less memory is required if all 256 possible pattern definitions are not required; the tables can be overlapped to reduce the amount of VRAM needed for pattern generation.

Sprites

The video display can have up to 32 sprites on the highest priority video planes. The sprites are special animation patterns which provide smooth motion and multilevel pattern overlaying. The location of a sprite is defined by the top left hand corner of the sprite pattern. The sprite can be easily moved pixel by pixel by redefining the sprite origin. This provides a simple but powerful method of quickly and smoothly moving special patterns. The sprites are not active in the Text mode. The 32 Sprite Planes are fully transparent outside of the sprite itself.

The sub-blocks in VRAM that define sprites are the Sprite Attribute Table and the Sprite Generator Table.

These tables are similar to their equivalents in the pattern realm in that the Sprite Attribute Table specifies where the sprite appears on the screen, while the Sprite Generator Table describes what the sprite looks like. Sprite Pattern formats are given in Table 5.

Since there are 32 sprites available for display, there are 32 entires in the Sprite Attribute Table. Each entry consists of four bytes. The entries are ordered so that the first entry corresponds to the sprite on the sprite 0 plane, the next to the sprite on the sprite 1 plane, and so on. The Sprite Attribute Table is 4*32 = 128 bytes long. The Sprite Attribute Table is located in a contiguous 128-byte block in VRAM beginning on a 128-byte boundary. The starting address of the Attribute Table is determined by the Sprite Attribute Table base address in VDP register 5. The base address forms the upper seven bits of the 14 bit VRAM address. The next 5 bits of the VRAM address are equal to the sprite number. The lowest 2 bits select one of the four bytes in the Attribute Table entry for each sprite. Each Sprite Attribute Table entry contains four bytes which specify the sprite position, sprite pattern name, and colour.

TABLE 6 Sprite pattern formats

SIZE	MAG	AREA	RESOLUTION	BYTES/PATTERN
0	0	8x8	single pixel	8
1	0	16x16	single pixel	32
0	1	16x16	2x2 pixels	8
1	1	32x32	2x2 pixels	32

The first two bytes of each entry of the Sprite Attribute Table determine the position of the sprite on the display. The first byte indicates the vertical distance of the sprite from the top of the screen, in pixels. It is defined such that a value of -1 puts the sprite butted up at the top of the screen, touching the backdrop area. The second bytes describes the horizontal displacement of the sprite from the left edge of the display. A value of 0 butts the sprite up against the left edge of the backdrop. Note that it is from the upper left pixel of the sprite that all measurements are taken.

When the first two bytes of an entry position of a sprite are overlapping the backdrop, the part of the sprite that is within the backdrop is displayed normally. The part of the sprite that overlaps the backdrop is hidden from view by the backdrop. This allows the animator to move a sprite into the display from behind the backdrop. The displacement in the first byte is partially signed, in that values for vertical displacement between -31 and 0 (E116 to 0) allow a sprite to "bleed in" from the top edge of the backdrop. Likewise, values in the range of 207 to 191 allow the sprite to bleed in from the bottom edge of the backdrop. Similarly, horizontal displacement values in the vicinity of 255 allow a sprite to bleed in from the right side of the screen. To allow sprites to bleed in from the left edge of the backdrop, a special bit in the third byte of the Sprite Attribute Table entry is used, as described in a later paragraph.

Byte 3 of the Sprite Attribute Table entry contains the pointer to the Sprite Generator Table that specifies what the sprite should look like. This is an 8-bit pointer to the sprite patterns definition, the Sprite Generator Table. The sprite name is similar to that in the Patterns Graphic mode.

Byte 4 of the Sprite Attribute Table entry contains the colour of the sprite in its lower 4 bits (see Table 2 for colour codes). The most significant bit is the Early Clock bit (EC). This bit, when set to a '0', does nothing. When set to '1', the horizontal position of the sprite is shifted to the left by 32 pixels. This allows a sprite to bleed in from the left edge of the backdrop. Values for horizontal displacement (byte 2 in the entry) in the range 0 to 32 cause the sprite to overlap with the left hand border of the backdrop.

The Sprite Generator Table is a maximum of 2048 bytes long beginning on the 2 kilobyte boundaries. It is arranged into 256 blocks of 8 bytes each. The third byte of the Sprite Attribute Table entry, then, specifies which eight byte block to use to specify a sprite's shape. The '1's in the Sprite Generator cause the sprite to be defined at that point; '0's cause the transparent colour to be used. The starting address of the table is determined by the sprite generator base address in the VDP register 6. The base address forms the 3 most significant bits of the 14-bit VRAM address. The next 8 bits of the address are equal to sprite name, and the last 3 bits are equal to the row number within the sprite pattern. The address formation is slightly modified for SIZE 1 sprites.

There is a maximum limit of four sprites that can be displayed on one horizontal line. If this rule is violated, the four highest-priority sprites on the line are displayed normally. The fifth and subsequent sprites are not displayed on that line. Furthermore, the fifth sprite bit in the VDP status register is set to a '1', and the number of the violating fifth sprite is loaded into the status register.

Larger sprites than 8x8 pixels can be used if desired. The MAG and SIZE bits in VDP register 1 are used to select the various options. The options are described here:

MAG=0,SIZE=0: No options chosen
MAG=1,SIZE=0: Eight bytes are still used in the Sprite Generator Table to describe the sprite; however, each bit in the Sprite Generator maps into 2 x 2 pixels on the TV screen, effectively doubling the size of the sprite to 16 x 16.
MAG=0,SIZE=1: 31 bytes are used in the Sprite Generator Table to define the sprite shape; the result is a 16 x 16 pixel sprite. Mapping is still one-bit-to-one pixel.
MAG=1,SIZE=1: Same as MAG=0,SIZE=1 except each bit now maps into a 2 x 2 pixel area, yielding a 32 x 32 sprite.

The VDP provides sprite coincidence checking. The coincidence status flag in the VDP status register is set to a '1' whenever two active sprites have '1' bits at the same screen location.

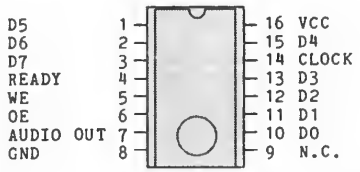
Sprite processing is terminated if the VDP finds a value of 208 (DO16) in the vertical position field of any entry in the Sprite Attribute Table. This permits the Sprite Attribute Table to be shortened to the minimum size required; it also permits the user to blank out part or all of the sprites by simply changing one byte in VRAM.

A total of 2176 VRAM bytes are required for the Sprite Name and Pattern Generator Tables. Significantly less memory is required if all 256 possible sprite pattern definitions are not required. The Sprite Attribute Table can also be shortened as described above. The tables can be overlapped to reduce the amount of VRAM required for sprite generation.

The sound processor used in the MTX500 Series computers is the Texas Instruments SN76489A sound generator IC. This device is I/O mapped as follows:

Data is mapped to output port 6
Strobe line is mapped to input port 3

To write data to the device send valid data to output port 5 and then strobe the data into the device by performing a dummy read from input port 3. The time interval between successive reads must be at least 32 clock cycles (32 T-states).



DESCRIPTION

The SN76489A digital complex sound generator is an I^2 L/Bipolar IC designed to provide low cost tone/noise generation capability in microprocessor systems. The SN76489A is a data bus based I/O peripheral.

RECOMMENDED OPERATING CONDITIONS

PARAMETER	MIN	TYP	MAX	UNITS
Supply Voltage, VCC	4.5	5.0	5.5	V
High Level Output Voltage, VOH (pin 4)			5.5	V
Low Level Output Current, IOL (pin 4)			2	mA
Operating Free-Air Temperature,TA	0		70	°C

OPERATION

1 Tone Generators

Each tone generator consists of a frequency synthesis and an attenuation section. The frequency synthesis section requires 10 bits of information (F9-F0) to define half the period of the desired frequency (n). F9 is the most significant bit and F0 is the least significant bit. This information is loaded into a 10 stage tone counter, which is decremented at a N/16 rate where N is the input clock frequency. When the tone counter decrements to zero, a borrow signal is produced. This borrow signal toggles the frequency flip-flop and also reloads the tone counter. Thus, the period of the desired frequency is twice the value of the period register.

The frequency can be calculated by the following:

$$f = \frac{N}{32n}$$

where N = ref clock in Hz
n = 10 bit binary number

The output of the frequency flip-flop feeds into a four stage attenuator. The attenuator values, along with their bit position in the data word, are shown in Table 1. Multiple attentuation control bits may be true simultaneousty. Thus, the maximum attenuation is 28 db.

TABLE 1 Attenuation Control

BIT POSITION				
A3	A2	A1	A0	WEIGHT
0	0	0	1	2 db
0	0	1	0	4 db
0	1	0	0	8 db
1	0	0	0	16 db
1	1	1	1	OFF

2 Noise Generator

The Noise Generator consists of a noise source and an attenuator. The noise source is a shift register with an exclusive OR feedback network. The feedback network has provisions to protect the shift register from being locked in the zero state.

TABLE 2 Noise Feedback Control

FB	CONFIGURATION
0	"Periodic" Noise
1	"White" Noise

Whenever the noise control register is changed, the shift register is cleared. The shift register will shift at one of four rates as determined by the two NF bits. The fixed shift rates are derived from the input clock.

TABLE 3 Noise Generator Frequency Control

BITS		
NF1	NF0	SHIFT RATE
0	0	N/512
0	1	N/1024
1	0	N/2048
1	1	Tone Generator Channel 3 Output

The output of the noise source is connected to a programmable attenuator as shown in figure 4.

3 Output Buffer/Amplifier

The output buffer is a conventional operational amplifier summing circuit. It sums the three tone generator outputs, and the noise generator output. The output buffer will generate up to 10mA.

4 CPU to SN76489A Interface

The microprocessor interfaces with the SN76489A by means of the 8 data lines and 3 control lines (WE, CE and READY). Each tone generator requires 10 bits of information to select the frequency and 4 bits of information to select the attenuation. A frequency update requires a double byte transfer, while an attenuator update requires a single byte transfer.

If no other control registers on the chip are accessed, a tone generator may be rapidly updated by initially sending both bytes of frequency and register data, followed by just the second byte of data for succeeding values. The register address is latched on the hip, so the data will continue going into the same register. This allows the 6 most significant bits to be quickly modified for frequency sweeps.

5 Control Registers

The SN76489A has 8 internal registers which are used to control the 3 tone generators and the noise source. During all data transfers to the SN76489A, the first byte contains a three bit field which determines the destination control register. The register address codes are shown in Table 4.

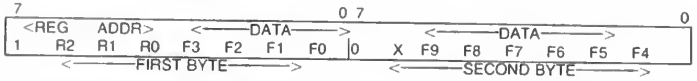
TABLE 4 Register Address Field

R2	R1	R0	DESTINATION CONTROL REGISTER
0	0	0	TONE 1 FREQUENCY
0	0	1	TONE 1 ATTENUATION
0	1	0	TONE 2 FREQUENCY
0	1	1	TONE 2 ATTENUATION
1	0	0	TONE 3 FREQUENCY
1	0	1	TONE 3 ATTENUATION
1	1	0	NOISE CONTROL
1	1	1	NOISE ATTENUATION

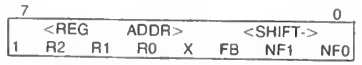
6 Data Formats

The formats required to transfer data are shown below.

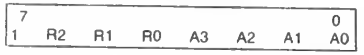
Update Frequency (Two Byte Transfer)



Update Noise Source (Single Byte Transfer)



Update Attenuator (Single Byte Transfer)



<REG ADDR> <DATA>

7 Data Formats

The microprocessor selects the SN76489A by placing CE into the true state (low voltage). Unless CE is true, no data can occur. When CE is true, the WE signal strobes the contents of the data bus to the appropriate control register. The data bus contents must be valid at this time.

The SN76489A requires approximately 32 clock cycles to load the data into the control register. The open collector READY output is used to synchronize the microprocessor to this transfer and is pulled to the false state (low voltage) immediately following the leading edge of CE. It is released to go to the true statement (external pullup) when the data transfer is completed. The data transfer timing is shown below.

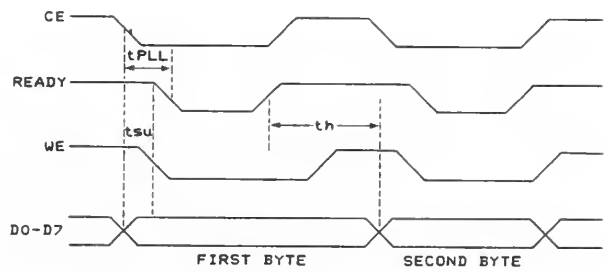
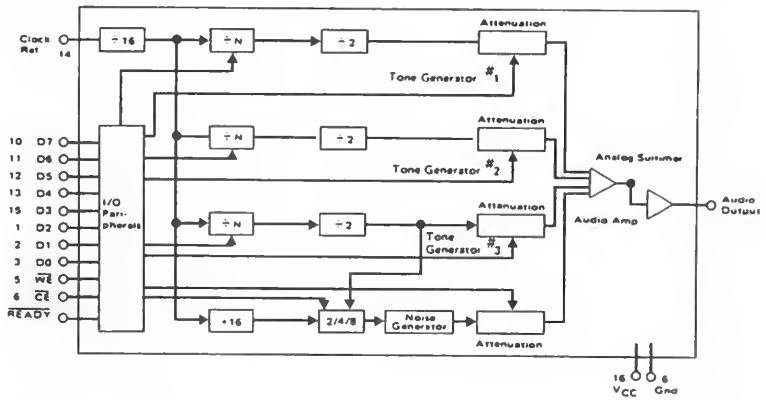


TABLE 5 Function Table

INPUTS		OUTPUT
CE	WE	READY
L	L	L
L	H	L
H	L	H
H	H	H

BLOCK DIAGRAM DESCRIPTION

This device consists of three programmable tone generators, a programmable noise generator, a clock scaler, individual generator attentuators and an audio summer output buffer. The SN76489A has a parallel 8 bit interface through which the microprocessor transfers the data which controls the audio output.



The paged memory map structure of the MTX Series computers has been designed to operate in two modes.

1 ROM BASED (RELCPMH = 0)

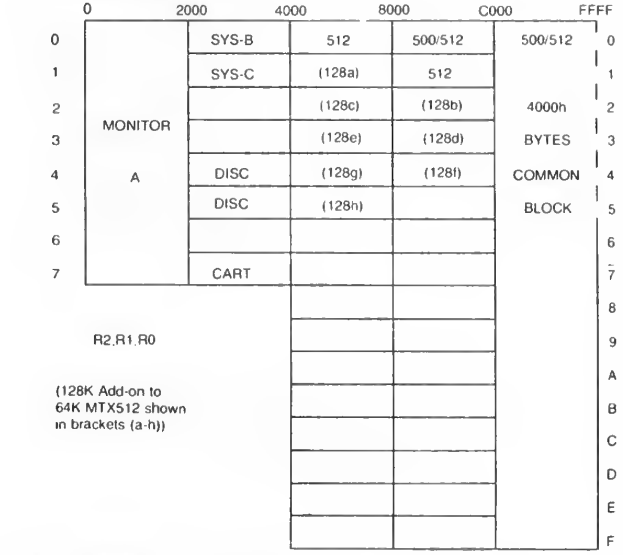
ROMs are mapped from 0 to 3FFFh. The 8K (2000h bytes) monitor ROM is always available in area 0 to 1FFFh and the paged ROMs of 8K (200h bytes) each are mapped from 2000h to 3FFFh as eight pages 0 to 7 set by R2,R1,R0 in the page port write only register. Up to 512K of RAM is mapped on 16 pages (0 to F) set up by P3,P2,P1 and P0 in the page port write only register. The area C000h to FFFFh is a 16K (4000h bytes) block common to all RAM pages. The 32K (8000h bytes) block from 4000h to BFFFh is mapped as 16 pages. The 32K bytes of RAM for an MTX500 is mapped from 8000h to FFFFh (page 0). The 64K bytes of RAM for an MTX512 is mapped from 4000h to FFFFh (page 0). The additional 16K is mapped from 8000h to C000h on page 1.

2 RAM BASED (RELCPMH = 1)

All ROMs are switched out in this mode, and up to 16 pages of 48K (C000h bytes) are mapped from 0 to BFFFh. These pages are set by P3,P2,P1 and P0 in the write only page port register. In the area C000h to FFFFh is a 16K block (4000h bytes) of RAM common to all pages.

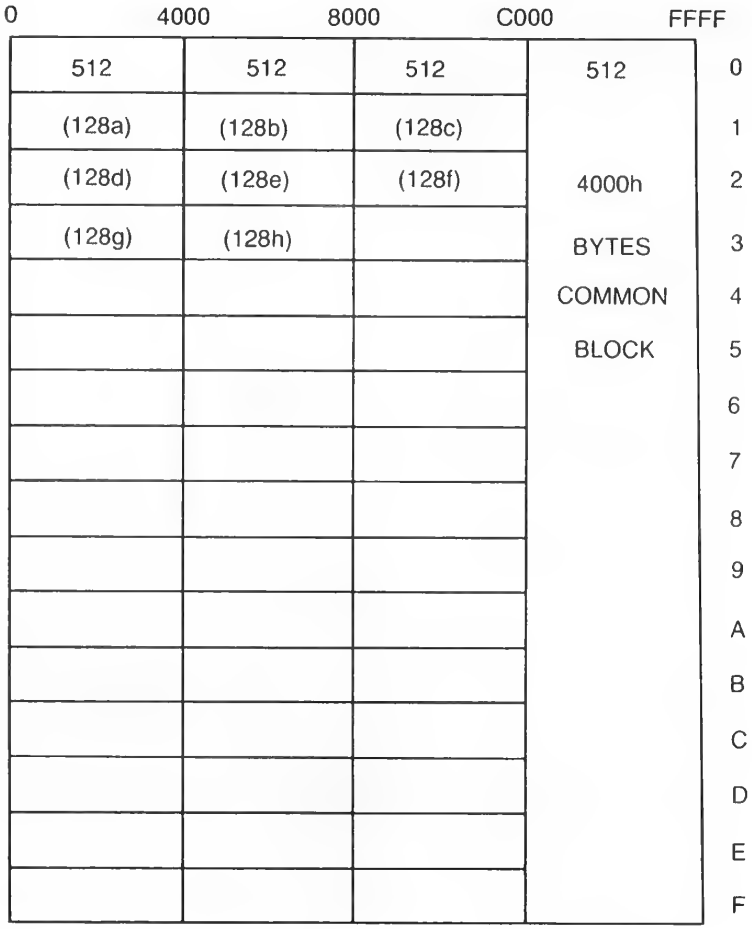
D7	D6	D5	D4	D3	D2	D1	D0
RELCPMH	R2	R1	R0	P3	P2	P1	P0

Write only page port register, output port 0.



ROM BASED MEMORY MAP RELCPMH = 0

P3,P2,P1,P0



RAM BASED MEMORY MAP RELCPMH 1
(128K Add-on to 64K MTX512 shown in brackets (a-h))

P3,P2,P1,P0

This section describes the MTX Series Port Map

00h
INPUT
IN(0) is used to set the printer STROBE (active low) to LOW. The STROBE line is reset HIGH either on CPU RESET or by IN(4). In the event of interrupt while STROBE is low it would be good practice to reset STROBE within an interrupt routine extending over a period of more than a few microseconds.

OUTPUT
OUT (0), d defines memory page address. The bit map is as follows:

D0 = P0
D1 = P1
D2 = P2
D3 = P3
D4 = R0
D5 = R1
D6 = R2
D7 = RELCPMH

Where the nibble P(i) defines the RAM page address, the 3 bit R(i) defines the ROM page address and bit 7 defines a ROM based system (D7 = 0) or a RAM based system (D7 = 1). The latch is reset to 0 on CPU reset.

01h
INPUT
IN(1),d VDP read (mode = 0) together with port 02 provide two contiguous read/write ports for the VDP. See documentation on the TMS9918 Series. Note Z80 CPU address line A1 is connected to mode input.

OUTPUT
OUT(1),d VDP write (mode = 0).

02h
INPUT
IN(2),d VDP read (mode = 1)

OUTPUT
OUT(2),d VDP write (mode = 1)

03h
INPUT
IN(3) This line is used as an output strobe into the sound generator. After data has been latched into the output port (6) data may be immediately strobed in using this line. A total of at least 32 clock cycles must have elapsed before additional data may be strobed in using IN(3).

OUTPUT
OUT(3),d This is the cassette output serial line. Valid data is placed on D0. This data bit is latched and appears on the cassette output (MIC) after attenuation (-20dB*VCC) and low pass filtering.

04h
INPUT
IN(4),d This is a nibble port for monitoring the status of the Centronics type parallel printer port.

D0 = BUSY active high handshake line
D1 = Error active low
D2 = PE paper empty active high
D3 = SLCT printer in selected state active high

OUTPUT
OUT (4),d Parallel 8 bit printer data. Valid data should be latched into this port. When status on IN(4) reads not BUSY and selected, then data should be strobed after a delay of approximately 1 microsecond using IN(0) to force STROBE low. After a further delay of approximately 1 microsecond STROBE should be forced high using IN(4).

05h
INPUT
IN(5),d This port is used to read the least significant 8 bits from the ten bit sense line of the 8x10 keyboard matrix.

OUTPUT
OUT(5),d This latched port provides the 8 drive lines of the 8x10 keyboard matrix.

06h
INPUT
IN(6),d This port is used to read in the two most significant sense lines (D0 and D1) of the 8x10 keyboard matrix. The two bit country code switch is read on D2 and D3.

OUTPUT
OUT(6),d This port is used to provide latched data for the sound generator which is subsequently strobed using IN(3).

07h
INPUT
IN(7),d This is the input port for the uncommitted parallel input output port (PI0). Data may be latched in for reading with an active low pulse on the enable line, designated INSTB.

OUTPUT
OUT(7),d This is the output port of the PI0. It is a latched output with tri-state output control using OTSTB.

08,09,0A,0Bh
These are four contiguous read/write ports for the four channels of the Z80A CTC.

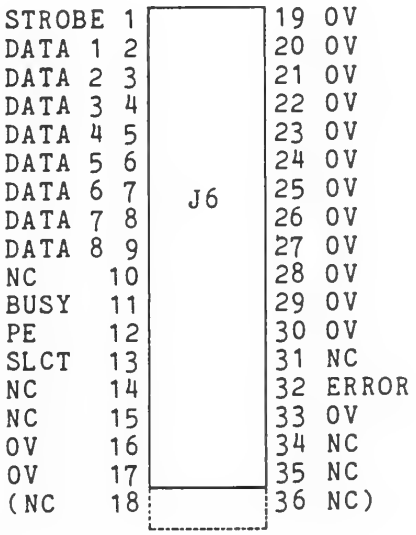
08 ch0 input-VDPINT	out-no connect
09 ch1 input-4MHz/13	out-DART ser clock 0
0A ch2 input-4MHz/13	out-DART ser clock 1
0B ch3 input-CSTTE edge	out-none

0C,0D,0E,0Fh
These are four contiguous read/write ports for the DART.

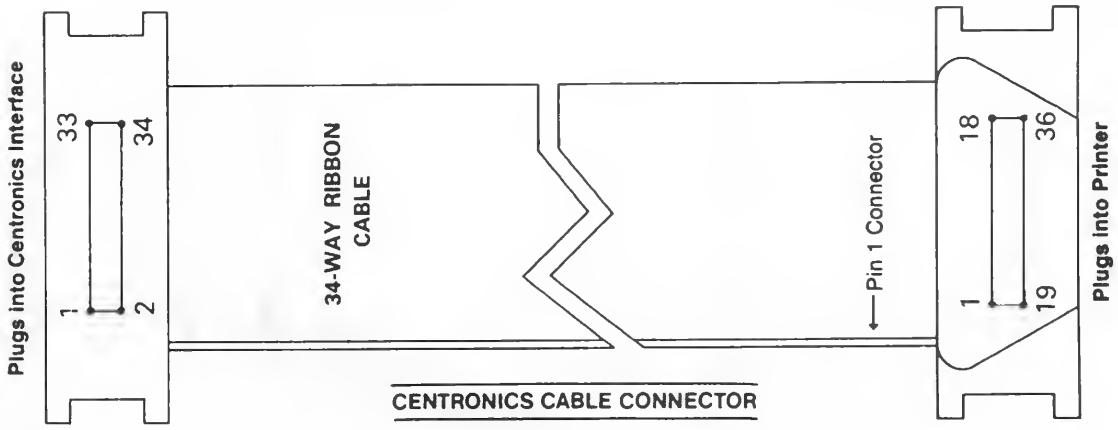
0C chA data
0DchB data
0E chA control
0F chB control

Ports 10h to 1Eh are currently unused with 1Fh reserved for cassette remote control.
Port addresses 20h upwards will be available as off-board I-0 ports in the disc expansion units.

APPENDIX 11 PARALLEL PRINTER INTERFACE

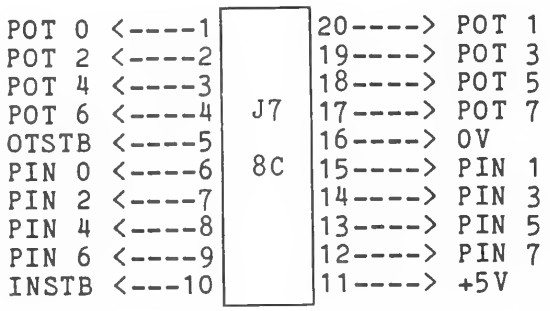


MTX500 Series Centronics Type Parallel Printer I/F Connector
34-Way (17+17) Right Angle Header Plug



APPENDIX 12 PARALLEL INPUT/OUTPUT PORT

This is an uncommitted TTL compatible PI0 and uses port 7, and is available on an internal 20 pin DIL socket. The port is normally transparent but input data may be latched by taking INSTB to a logic low. The output port is normally tri-state but may be made active by taking OTSB to a logic low. Only TTL compatible signals may be used. The 5V current drain must not exceed 20mA.



13 MEMOTECH DMX80 PARALLEL PRINTER CONNECTOR

SIGNAL PIN No.	RETURN PIN No.	SIGNAL	DIRECTION
1	19	STROBE	IN
2	20	DATA 1	IN
3	21	DATA 2	IN
4	22	DATA 3	IN
5	23	DATA 4	IN
6	24	DATA 5	IN
7	25	DATA 6	IN
8	26	DATA 7	IN
9	27	DATA 8	IN
10	28	ACKNLG	OUT
11	29	BUSY	OUT
12	30	PE	OUT
13	-	SLCT	OUT
14	-	AUTO FEED XT	IN
15	-	NC	-
16	-	CHASSIS-GND	-
18	-	NC	-
19-30	-	GND	-
31	-	INIT	IN
32	-	ERROR	OUT
33	-	GND	-
34	-	NC	-
35	-	-	-
36	-	SLCT-IN	IN

COMMAND		Control Code	INPUT FORMAT
CARRIAGE RETURN		CR	LPRINT CHR$(13);
LINE FEED		LF	LPRINT CHR$(10);
VERTICAL TABULATION		VT	LPRINT CHR$(11);
FORM FEED		FF	LPRINT CHR$(12);
HORIZONTAL TABULATION		HT	LPRINT CHR$(9);
SHIFT OUT		SO	LPRINT CHR$(14);
SHIFT IN		SI	LPRINT CHR$(15);
DEVICE CONTROL 1		DC1	LPRINT CHR$(17);
DEVICE CONTROL 2		DC2	LPRINT CHR$(18);
DEVICE CONTROL 3		DC3	LPRINT CHR$(19);
DEVICE CONTROL 4		DC4	LPRINT CHR$(20);
ESCAPE		ESC	LPRINT CHR$(27);
BACK SPACE		BS	LPRINT CHR$(8);
DELETE		DEL	LPRINT CHR$(127);
BELL		BEL	LPRINT CHR$(7);
NULL		NULL	LPRINT CHR$(0);
PICA PITCH DESIGNATION		ESC+P+(1)	LPRINT CHR$(27); "P" CHR$(1);
ELITE PITCH DESIGNATION		ESC+P+(0)	LPRINT CHR$(27); "P"; CHR$(0);
DOUBLE WIDTH ELONGATED CHARACTER DESIGNATION	SETTING	ESC+W+(1)	LPRINT CHR$(27); "W"; CHR$(1);
	RELEASE	ESC+W+(0)	LPRINT CHR$(27); "W" CHR$(0);
EMPHASIZED CHARACTER DESIGNATION	SETTING	ESC+E	LPRINT CHR$(27); "E";
	RELEASE	ESC+F	LPRINT CHR$(27); "F";
DOUBLE PRINTED CHARACTER DESIGNATION	SETTING	ESC+G	LPRINT CHR$(27); "G";
	RELEASE	ESC+H	LPRINT CHR$(27); "H";
UNDERLINE DESIGNATION	SETTING	ESC+-+(1)	LPRINT CHR$(27), "-", CHR$(1),
	RELEASE	ESC+-+(0)	LPRINT CHR$(27); "-"; CHR$(0);
SUPERSCRIPT DESIGNATION		ESC+S+(0)	LPRINT CHR$(27); "S"; CHR$(0);
SUBSCRIPT DESIGNATION		ESC+S+(1)	LPRINT CHR$(27); "S"; CHR$(1);
SUPERSCRIPT/SUBSCRIPT RELEASE		ESC+T	LPRINT CHR$(27); "T";
ITALIC FONT DESIGNATION	SETTING	ESC+4	LPRINT CHR$(27); "4";
	RELEASE	ESC+5	LPRINT CHR$(27); "5";

COMMAND		Control Code	INPUT FORMAT
FORM LENGTH DESIGNATION (LINE NUMBER)		ESC+C+(n)	LPRINT CHR$(27); "C"; CHR$(n);
FORM LENGTH DESIGNATION (INCH NUMBER)		ESC+C+(0)+(m)	LPRINT CHR$(27); "C"; CHR$(0); CHR$(m);
LINE SPACING PITCH DESIGNATION	1/8 INCH FEED	ESC+0	LPRINT CHR$(27), "0";
	7/72 INCH FEED	ESC+1	LPRINT CHR$(27); "1";
	1/6 INCH FEED	ESC+2	LPRINT CHR$(27); "2";
	n/216 INCH FEED	ESC+3+(n)	LPRINT CHR$(27); "3"; CHR$(n);
	n/72 INCH FEED	ESC+A+(n)	LPRINT CHR$(27), "A"; CHR$(n);
SINGLE NEW LINE DESIGNATION		ESC+J+(n)	LPRINT CHR$(27); "J"; CHR$(n).
PAPER END SIGNAL IGNORE	SETTING	ESC+8	LPRINT CHR$(27); "8";
	RELEASE	ESC+9	LPRINT CHR$(27); "9";
SKIP PERFORATION	SETTING	ESC+N+(n)	LPRINT CHR$(27); "N"; CHR$(n);
	RELEASE	ESC+O	LPRINT CHR$(27); "O";
VT	SETTING	ESC+B+(n1)+(n2)+...+(nx)+(0)	LPRINT CHR$(27); "B"; CHR$(n1); CHR$(n2), ...CHR$(nx); CHR$(0);
	RELEASE	ESC+B+(0)	LPRINT CHR$(27); "B"; CHR$(0);
PRINTING WIDTH DESIGNATION		ESC+O+(n)	LPRINT CHR$(27); "O"; CHR$(n);
HT	SETTING	ESC+D+(n1)+(n2)+...+(nx)+(0)	LPRINT CHR$(27); "D"; CHR$(n1); CHR$(n2);...CHR$(nx); CHR$(0);
	RELEASE	ESC+D+(0)	LPRINT CHR$(27); "D"; CHR$(0);
SINGLE-DIRECTION PRINTING DESIGNATION	SETTING	ESC+U+(1)	LPRINT CHR$(27); "U"; CHR$(1);
	RELEASE	ESC+U+(0)	LPRINT CHR$(27); "U"; CHR$(0);
INTERNATIONAL CHARACTER SELECTION		ESC+R+(n)	LPRINT CHR$(27); "R"; CHR$(n);
STANDARD BIT IMAGE DESIGNATION		ESC+K+(n1)+(n2)	LPRINT CHR$(27); "K"; CHR$(n1); CHR$(n2);
DOUBLE DENSITY BIT IMAGE DESIGNATION		ESC+L+(n1)+(n2)	LPRINT CHR$(27); "L"; CHR$(n1); CHR$(n2);
FONT REGISTRATION	SETTING	ESC+Y+hexa code+(D1)+(D2)+...+(D9)	LPRINT CHR$(27); "Y"; hexa code; CHR$(D1); CHR$(D2) +. .CHR$(D9);
	RELEASE	ESC+Z+hexa code	LPRINT CHR$(27); "Z"; hexa code;
MSB OPERATION	SETTING	ESC+>	LPRINT CHR$(27); ">";
	RESETTING	ESC+=	LPRINT CHR$(27); "=";
	CANCELING	ESC+#	LPRINT CHR$(27); "#";
ONE LINE SINGLE-DIRECTIONAL PRINTING		ESC+<	LPRINT CHR$(27); "<";
RESET PRINTER		ESC+@	LPRINT CHR$(27); "@";

APPENDIX 14 PAL LISTINGS

```
PAL14L4   PAL DESIGN SPECIFICATION
MTX500 FPLA      GEOFF BOYD 05JUN83
MEMORY SEGMENT DECODER FOR 16K+8KROM/32K+512KRAM
MEMOTECH LTD  3 COLLINS ST OXFORD
A13 A14 A15 P2 R1 R2 P1 MREQL RDL GND
I2H4L PO RELCPMH RAM NA15 CE64 CEA P3 RO VCC
/CEA = /RELCPMH*/A15*/A14*/A13*/MREQL*/RDL
      + /RELCPMH*/A15*/A14*A13*/R2*/R1*/RO*/MREQL*/RDL
/CE64 = /RELCPMH*/R2*/R1*RO*/A15*/A14*A13*/MREQL*/RDL
        +/RELCPMH*R2*R1*RO*/A15*/A14*A13*/MREQL*/RDL
/NA15 = /A15 + /P3*/P2*/P1*PO*/RELCPMH*/A14*A15
/RAM = /P3*/P2*/P1*/PO*/I2H4L*RELCPMH*/A14*/MREQL
       + /P3*/P2*/P1*/PO*/A14*A15*/MREQL
       + A14*A15*/MREQL
       + /P3*/P2*/P1*PO*/RELCPMH*/A14*A15*/MREQL*/I2H4L
DESCIPTION: MTX DECODING FOR 1*16K+1*8KROMS.
END.


PAL14L4   PAL DESIGN SPECIFICATION
MTX512 FPLA          GEOFF BOYD 05JUN83
MEMORY SEGMENT DECODER 16+8K ROM 64K RAM
MEMOTECH LTD  3 COLLINS ST OXFORD
A13 A14 A15 P2 R1 R2 P1 MREQL RDL GND
I2H4L PO RELCPMH RAM NA15 CE64 CEA P3 RO VCC
/CEA =/RELCPMH*/A15*/A14*/A13*/MREQL*/RDL
       +/RELCPMH*/R2*/R1*/RO*/A15*/A14*A13*/MREQL*/RDL
/CE64 = /RELCPMH*/R2*/R1*RO*/A15*/A14*A13*/MREQL*/RDL
        +/RELCPMH*R2*R1*RO*/A15*/A14*A13*/MREQL*/RDL
/NA15 = /A15 + /P3*/P2*/P1*PO*/RELCPMH*/A14*A15
/RAM = /P3*/P2*/P1*/PO*/I2H4L*RELCPMH*/MREQL
       + /P3*/P2*/P1*/PO*A14*/A15*/MREQL*/I2H4L*/RELCPMH
       + A14*A15*/MREQL
       + /P3*/P2*/P1**/A14*A15*/MREQL*/I2H4L*/RELCPMH
DESCRIPTION :
END.


PAL14L4 PAL DESIGN SPEC
MTX500-RS232   GEOFF BOYD  01JULY83
RS232 AND I/F DECODER
MEMOTECH LTD    3 COLLINS ST  OXFORD
MREQL RDL MIL IORQL DTIEO BUSAKL A7 A6 A5 GND
A432 RELCPMH R1 245DIR DARTEN 03 ND R2 EXT245 VCC
/245DIR = /EXT245
       + /MIL*/IORQL*DTIEO
       + /BUSAKL*RDL
/ND = A7*/IORQL*/RDL*MIL
      + A6*/IORQL*/RDL*MIL
      + A5*/IORQL*/RDL*MIL
/03 = /MREQL*/RDL*/RELCPMH*/R1*R2
/DARTEN = /A7*/A6*/A5*/A432*/IORQL*MIL
DESCRIPTION :
```

```
PAL14L4   PAL DESIGN SPECIFICATION
MTX512 FPLA          GEOFF BOYD 05JUN83
MEMORY SEGMENT DECODER   3*8K ROM 64K RAM
MEMOTECH LTD  3 COLLINS ST OXFORD
A13 A14 A15 P2 R1 R2 P1 MREQL RDL GND
I2H4L PO RELCPMH RAM NA15 CE64 CEA P3 RO VCC
/CEA =/RELCPMH*/A15*/A14*/A13*/MREQL*/RDL
/CE64 = /RELCPMH*/R2*/R1*/A15*/A14*A13*/MREQL*/RDL
        +/RELCPMH*R2*R1*RO*/A15*/A14*A13*/MREQL*/RDL
/NA15 = /A15 + /P3*/P2*/P1*PO*/RELCPMH*/A14*A15
/RAM = /P3*/P2*/P1*/PO*/I2H4L*RELCPMH*/MREQL
       + /P3*/P2*/P1*/PO*A14*/A15*/MREQL*/I2H4L*/RELCPMH
       + A14*A15*/MREQL
       + /P3*/P2*/P1**/A14*A15*/MREQL*/I2H4L*/RELCPMH
DESCRIPTION:
END.
```

```
PAL12L6     PALDESIGN SPECIFICATION
MTX500 SERIES ROM EXT PAK GEOFF BOYD 15FEB84
MEMOTECH
MEMOTECH STATION LANE WITNEY
BTRST RELCPMH R2 R1 RO QO Q1 WRL MREQL GND
A13 A14 CEO CE3 NC OS CE1 CE2 A15 VCC
/OS = /A15*/A14*/A13*/MREQL*/WRL*/RELCPMH
/CEO = /A15*/A14*A13*/MREQL*/RELCPMH*/R2*R1*/RO*/Q1*/QO
/CE1 = /A15*/A14*A13*/MREQL*/RELCPMH*/R2*R1*/RO*/Q1*QO
/CE2 = /A15*/A14*A13*/MREQL*/RELCPMH*/R2*R1*/RO*Q1*/QO
/CE3 = /A15*/A14*A13*/MREQL*/RELCPMH*/R2*R1*/RO*Q1*QO
DESCRIPTION:
THE ABOVE DECODES 4 8K ROMS 2764 TO PAGE 2 SUBPAGES 0 TO 3
A MEMORY  WRITE TO 0 TO 8K IN ROMBASE SELECTS SUB PAGE
0,1,2,3 SET BY DO,D1.CLEARS ON RESET TO SUBPAGE 0
A TOTAL OF 4MEGABYTES OF ROM SPACE IS AVAILABLE ON PAGES 2 AND 3
```

```
PAL14L4   PAL DESIGN SPECIFICATION
MTX500 FPLA          GEOFF BOYD 05JUN83
MEMORY SEGMENT DECODER FOR 3*8K ROM 32K RAM
MEMOTECH LTD  3 COLLINS ST OXFORD
A13 A14 A15 P2 R1 R2 P1 MREQL RDL GND
I2H4L PO RELCPMH RAM NA15 CE64 CEA P3 RO VCC
/CEA =/RELCPMH*/A15*/A14*/A13*/MREQL*/RDL
/CE64 = /RELCPMH*/R2*/R1*/A15*/A14*A13*/MREQL*/RDL
        +/RELCPMH*R2*R1*RO*/A15*/A14*A13*MREQL*/RDL
/NA15 = /A15 + /P3*/P2*/P1*PO*/RELCPMH*/A14*A15
/RAM = /P3*/P2*/P1*/PO*/I2H4L*RELCPMH*/A14*/MREQL
       + /P3*/P2*/P1*/PO*/A14*A15*/MREQL
       + A14*A15*/MREQL
       + /P3*/P2*/P1*PO*/RELCPMH*/A14*A15*/MREQL

DESCRIPTION:
```

INDEX

```
100 VS 4: COLOUR 4,1: COLOUR 2,1
200 CTLSPR 0,1: CTLSPR 2,32: CTL
300 GENPAT 3,1,60,60,24,126,126,
400 FOR A=1 TO 32: SPRITE A,1,1
500 LET A=ASC(INKEY$)
600 IF A=8 THEN VIEW 4,5 ELSE
700 IF A=11 THEN VIEW 6,5 ELSE
800 IF A=26 THEN GOTO 400
900 GOTO 500
```

MEMOTECH

Memotech Limited, Witney, Oxon OX8 6BX U.K.

Memotech Corporation, 99 Cabot Street, Needham, MA 02194 U.S.A.